TURING 图灵程序设计丛书

MANNING

Mahout in Action

Mahout实战

[美] Sean Owen Robin Anil Ted Dunning Ellen Friedman 著

王斌 韩冀中 万吉 译

Apache基金会官方推荐

Mahout核心团队权威力作

大数据时代机器学习的实战经典



人民邮电出版社
POSTS & TELECOM PRESS



Mahout in Action

Mahout实战

"全面介绍Mahout机器学习实战的佳作。"

——Isabel Drost，Apache Mahout创始人

"深入浅出，复杂概念都讲解得透彻明白。"

——Rick Wagner，Red Hat

"出自核心开发团队之手，学习Mahout必读。"

——Philipp K. Janert，*Gnuplot in Action* 作者

通过收集数据来学习和演进的计算机系统威力无穷。Mahout作为Apache的开源机器学习项目，把推荐系统、分类和聚类等领域的核心算法浓缩到了可扩展的现成的库中。使用Mahout，你可以立即在自己的项目中应用亚马逊、Netflix及其他互联网公司所采用的机器学习技术。

本书出自Mahout核心成员之手，得到Apache官方推荐，权威性毋庸置疑。作者凭借多年实战经验，为读者展现了丰富的应用案例，并细致地介绍了Mahout的解决之道。本书还重点讨论了可扩展性问题，介绍了如何利用Apache Hadoop框架应对大数据的挑战。

本书内容：

- 利用分组数据实现个性化推荐；
- 寻找数据中的逻辑簇；
- 通过即时分类实现过滤与调优。

MANNING

图灵社区：iTuring.cn

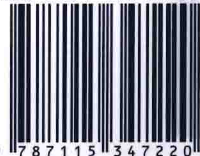
热线：(010)51095186转600

分类建议 计算机/程序设计/Mahout

人民邮电出版社网址：www.ptpress.com.cn







ISBN 978-7-115-34722-0

定价：79.00元

TURING 图灵程序设计丛书



Mahout in Action

Mahout 实战

[美] Sean Owen Robin Anil Ted Dunning Ellen Friedman 著

王斌 韩冀中 万吉 译

人民邮电出版社

北 京

图书在版编目（C I P）数据

Mahout实战 / (美) 欧文 (Owen, S.) 等著 ; 王斌, 韩冀中, 万吉译. -- 北京 : 人民邮电出版社, 2014.3
(图灵程序设计丛书)
书名原文: Mahout in action
ISBN 978-7-115-34722-0

Ⅰ. ①M… Ⅱ. ①欧… ②王… ③韩… ④万… Ⅲ. ①机器学习②电子计算机－算法理论 Ⅳ. ①TP181 ②TP301.6

中国版本图书馆CIP数据核字(2014)第031399号

内 容 提 要

本书是Mahout领域的权威著作，出自该项目核心成员之手，立足实践，全面介绍了基于Apache Mahout的机器学习技术。本书开篇从Mahout的故事讲起，接着分三部分探讨了推荐系统、聚类和分类，最后的附录涵盖JVM调优、Mahout数学知识和相关资源。

本书适合所有数据分析和数据挖掘人员阅读，需要有Java语言基础。

◆ 著　[美] Sean Owen　Robin Anil　Ted Dunning　Ellen Friedman
译　王　斌　韩冀中　万　吉
责任编辑　毛倩倩
执行编辑　刘　帅
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：800×1000　1/16
印张：21.25
字数：502千字　2014年3月第1版
印数：1－4 000册　2014年3月河北第1次印刷
著作权合同登记号　图字：01-2011-7805号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第0021号

前　言

追溯这本书的由来，就我个人而言，要从2005年说起。我的一个朋友当时正在创办一家公司，急需协同过滤技术。虽然当时可以找到成熟、开源的软件包，但是它们要么太过繁复，要么太学术化。所以，我决定从零开始，为这个朋友的创业公司开发了一个推荐系统的简单原型。遗憾的是，这家创业公司夭折了。然而，我却无法说服自己删除这个原型。它实在有趣，于是我对它进行整理并写了文档，用Taste这个名字将它发布为一个开源项目。

一年无声地过去了。我在业余时间为其增加了一些代码，并修复了一些问题。接着，有一两个用户出现，并提交了一些软件bug和补丁，然后又有了几个用户，再后来又增加了好一些。到了2008年，虽然小但却稳定的用户群形成了。后来，Apache Lucene的人把机器学习相关的部分剥离出来形成了Apache Mahout，他们建议把我们的两个项目进行合并。此后，本书在2009年晚些时候开始立项。而今，当看到这个项目滚雪球般地发展到2011年，并开始被大公司在生产系统中使用，我自己既惊讶又欣喜。

的确，我只是无心插柳。即便我已经是一个高级工程师，曾在谷歌工作过，也没有人会误认为我是这个领域的专家。我更像是一个博物馆的管理者，而不是一个画家，我将一个领域的伟大思想进行搜集、组织和打包，使之广为所用。这同样不失为一项有用的工作。

一些人在读过本书的初稿之后，说它是一本“通俗易懂”的机器学习书。这是一种盛誉，而我完全赞同。机器学习有其魔力所在，不过这个领域中有许多研究性的著作对于非专业人员而言就像天书，它们也与该技术的应用实践相去甚远。而本书旨在让读者易于理解，为爱好者揭示领悟的快乐，并为实践者节省工作时间。我希望你阅读本书时的惊喜比疑问多。

——Sean Owen

我对机器学习的兴趣可以回溯到2006年上大学的那段日子。那时，我作为实习生和一组人共同设计一个个性化的推荐引擎。这个小组后来成长为Minekey公司；我也被邀请加入，成为其核心开发人员。后来的四年，我一直从事机器学习技术的实现与试验。在此期间，我偶然间发现了Mahout，并开始作为一个Google Summer of Code的参赛学生加入这个项目。我记得，接下来的事情就是不断为它的代码库贡献算法和补丁，做性能调优，以及帮助邮件列表中的其他人。

由机器学习开发者、研究者和爱好者组成的社区非常出色，正在不断成长，而我有幸成为这个团队的一员。随着越来越多的公司采用Mahout，它正在成为机器学习的主流软件库。我衷心希望你在阅读本书时能够乐在其中。

——Robin Anil

我（Ted）在机器学习上是先做研究后做项目。我早期从事的是学术工作，后来参与了一些创业团队，从而得以将机器学习在实际中应用。

我（Ellen）以前在生物化学和分子生物学实验室工作。在研究大量数据的同时，我还写了许多技术文章。此番经历，让我痴迷于数据及其蕴含的意义。我努力把这种内在的东西写进本书里。

我们两个人一致认为开源有赖于一个有大量活跃用户参与的社区。Mahout的成功主要来自于那些使用这个软件的人，他们通过在邮件列表中展开讨论、修复bug，以及提供建议，把使用经验回馈到这个项目中。

为此，本书不仅给出代码的实用注解，而且引出了代码背后的一些概念。介绍隐藏于代码背后的框架，会使你更有效地加入到Mahout的讨论中，并从中获益。我们希望本书不仅能够帮助读者，而且能够使Mahout自身得到完善和发展。

——Ted Dunning和Ellen Friedman

致谢

本书的出版离不开众人的努力。作者对他们致以衷心的感谢，但限于篇幅，致谢名单只列出了其中一部分人，排名不分先后。

- 在机器学习领域发表核心文章的研究者，详见附录C。
- 花时间测试试用版软件的Mahout用户，他们寻找与解决bug，为软件打补丁乃至提出建议。
- Mahout提交者，他们致力于Mahout的发展、完善和提升。
- Manning出版社投入了大量时间和精力将本书出版并投入市场。特别感谢Katharine Osborne、Karen Tegtmeyer、Jeff Bleiel、Andy Carroll、Melody Dolab和Dottie Marsico，你看到的最终稿与他们的工作密不可分。
- 在本书写作过程中提供了宝贵反馈的审校者：Philipp K. Janert、Andrew Oswald、John Griffin、Justin Tyler Wiley、Deepak Vohra、Grant Ingersoll、Isabel Drost、Kenneth DeLong、Eric Raymond、David Grossman、Tom Morton，以及Rick Wagner。
- Alex Ott在本书印刷前一刻对全稿进行了细致的技术审核。
- 在作者在线论坛上发帖评价本书的MEAP读者[①]。
- 每个在Mahout邮件列表中提问的人。
- 在本书长时间写作过程之中，给予我们无尽支持的家人和朋友！

① MEAP，全称Manning Early Access Program。因为图书出版周期较长，为让读者可以时刻了解热门技术，Manning出版社推出了这一图书抢鲜阅读项目。参与其中的读者可以在图书未编辑完成之际，一章一章地阅读。——编者注

关 于 本 书

你可能还有疑问：这本书是否适合我？

如果你正在寻找一本机器学习教材，答案是否定的。本书不会对诸多算法和技术的理论以及推导过程给出全面解释。如果你了解机器学习技术，并熟悉矩阵和向量等相关的数学概念，这有助于阅读本书，但并非必要条件。

如果你正在开发先进、智能的应用，答案则是肯定的。本书从实践而非理论入手来诠释这些技术，并给出完整的例子与解决方案。它在教授用Mahout解决问题的同时，带给你实践者的经验与领悟。

如果你是人工智能、机器学习及相关领域的研究人员，答案亦是肯定的。你所面对的最大障碍，很可能就是把新的算法应用到实践中。对于新的大规模算法的测试与部署，Mahout提供了一个成熟的框架、一系列模式以及现成的组件。本书是你在复杂的分布式计算框架上学习开发机器学习系统的“快车票”。

如果你正领导一个产品小组或创业团队，想利用机器学习创造一种竞争优势，那么本书同样适合你。通过实际示例，它让你了解这些技术的多种应用方式。它还会让小型技术团队变得高效，能够处理大量数据，而这在以前只有具有大量技术资源的组织机构才做得到。

路线图

本书分为三部分，分别介绍了Apache Mahout中的协同过滤、聚类和分类。

首先，第1章整体介绍Apache Mahout。这一章为你阅读后续各章奠定基础。

第一部分（第2章~第6章）由Sean Owen编写，主要介绍协同过滤与推荐。第2章基于Mahout构造推荐引擎并评估其性能。第3章探讨如何高效呈现推荐引擎使用的数据。第4章介绍可在Mahout中利用的所有推荐算法，并比较它们的优缺点。在此背景之下，第5章给出一个案例，将第4章介绍的推荐系统实现应用在该案例的真实问题中，配以某些特定属性的数据，从而建立一个可为生产环境所用的推荐引擎。第6章介绍Apache Hadoop，通过研究基于Hadoop的推荐引擎，首次为你展现分布式环境中的机器学习算法。

第二部分（第7章~第12章）探索Apache Mahout上的聚类算法。通过Robin Anil所做的技术说明，你可以把看起来类似的数据片段组织为一个个集合或者说簇（cluster）。聚类有助于揭示大规模数据中有趣的信息组合。这部分从聚类中的简单问题开始介绍，并给出了Java示例。接下来，

作者引入更多实际示例，并展示如何让Apache Mahout以Hadoop作业的方式运行，从而轻而易举地聚类大量数据。

第三部分（第13章~第17章）由Ted Dunning和Ellen Friedman编写，探索如何用Mahout进行分类。首先，作者带你了解如何通过一组示例“教会”一个算法，从而建立和训练分类器模型。接下来，你会了解如何评估并微调分类器模型以得到更好的结果。这部分最后以一个分类实战案例结束。

代码约定及下载

本书源代码均采用等宽字体印刷，列为代码清单，并对重点进行注释。代码清单旨在简单明了，重点突出。它们通常不给出Java导入包、类声明、Java注释，以及其他对代码的讨论无关紧要的东西。

本书中的类名亦采用等宽字体，放于文本之间，以显示它们是可以在Apache Mahout源代码中找到并研究的类名。例如，`LogLikelihoodSimilarity`是Mahout中的一个Java类。

一些代码清单中列出了可执行命令。它们是为Mac OS X和Linux发行版等类Unix环境而写的。如果使用了类Unix的Cygwin环境，它们也可以在微软Windows系统下运行。

本书关键代码清单中的源代码均可编译，且均可从www.manning.com/MahoutinAction下载[①]。这些都是独立的Java源文件，并不包括编译脚本。为了方便起见，你可以把它们解压到Mahout源代码发布包的examples/src/java/main目录下。这样，Mahout的编译环境将会自动编译这些代码。

多媒体资料

四位作者均录制了音频和视频片段，与书中多数章中的特定节互相补充，为相应话题提供了附加信息。你可以从本书英文版电子书中看到或听到这些音视频片段，该电子书对英文纸质书的拥有者免费；你还可以从www.manning.com/MahoutinAction/extras免费获取。通过书中的音频和视频图标，你可以获知其所涉及的话题，以及发言者是谁。这些多媒体资料的清单详见“关于多媒体资料”。

作者在线

本书英文版读者可以免费访问Manning出版社专门维护的一个论坛，并可以发表评论、提出技术问题，并获得作者和其他论坛用户的帮助。你可以通过网页www.manning.com/MahoutinAction进入和订阅该论坛。完成注册后，你可以了解如何使用该论坛、该论坛所能提供的帮助，以及论坛的行为规范。

Manning出版社承诺为读者和作者提供一个进行深入对话的场所，但不对作者的参与程度做要求，他们对于该论坛的贡献是出于自愿且无报酬的。我们建议读者尽量向作者提一些具有挑战性的问题，让他们保持兴趣！

本书在印期间，读者均可访问作者在线论坛，并查看之前的讨论。

---

① 亦可在图灵社区（iTuring.cn）本书页面免费注册下载。——编者注

关于多媒体资料

本书附带的多媒体资料可以在www.manning.com/MahoutinAction/extras/上免费收听或收看。本书中空白处的音频或视频文件图标（如下所示），指出了书中哪些地方可参考附加资料。


Audio icon


Video icon

No. 1　音频　p2
Sean介绍了Mahout项目以及他参与的事项。

No. 2　音频　p19
Sean讨论了推荐系统的工作。

No. 3　音频　p29
Sean阐述为什么他认为人们有可能过度"聆听"数据。

No. 4　音频　p42
Sean谈论皮尔逊相关系数的实现。

No. 5　音频　p63
Sean讨论了诠释性能指标的价值。

No. 6　音频　p84
Sean解释了Mahout和Hadoop之间的关系。

No. 7　音频　p108
Robin解释了如何为一个数据集选择正确的距离测度方法。

No. 8　音频　p114
Robin扩展了苹果的类比示例。

No. 9　音频　p127
Robin解释了k-means聚类迭代过程。

No. 10　音频　p165
Robin讨论改善聚类质量的策略。

No. 11　音频　p179
Robin解释了如何改进大规模聚类的性能。

No. 12 视频 p208
Ellen展示了如何训练一个模型使之逐步优化。

No. 13 视频 p234
Ted和Ellen展示了Logistic回归的内部机制。

No. 14 视频 p238
Ted比较了使用串行算法与并行算法的优势。

No. 15 音频 p249
Ted和Ellen讨论了AUC评估方法。

No. 16 音频 p252
Ted和Ellen讨论了为什么对数似然法意味着“永不说不”。

关于封面

封面上是“一个来自Rakov-Potok的男人”。Rakov-Potok是克罗地亚北方的一个村庄。该图取自克罗地亚19世纪中叶传统服饰影集的一个副本，作者为Nikola Arsenovic，由Ethnographic博物馆在2003年出版于克罗地亚的斯普利特。该图得自于乐于助人的Ethnographic博物馆馆员，这个博物馆位于该城镇在中世纪罗马时的核心位置，是公元304年左右罗马皇帝戴克里先的宫殿遗址。这本书包含来自克罗地亚不同地域的颜色精美的插图，附有服饰和日常生活的说明。

Rakov-Potok是一个风景如画的乡村，位于Samobor山脚下、萨瓦河土地肥沃的河谷中，距Zagreb城不远。它有着悠久的历史，在那里，你会与许多城堡、教堂和中世纪甚至罗马时期的遗迹不期而遇。封面上的人物身着白色羊毛长裤和白色羊毛外套，上面有着大量的红色和蓝色绣花——这是该地区山区居民的典型装束。

过去200年间，人们的着装和生活方式已经发生变化，曾经如此丰富的地域多样性已渐渐消失了。现在，各大洲的居民已经很难分辨，更遑论不同小村或距离只有几英里的人。也许我们用文化多样性换来的是更多样化的个人生活——必然是更为丰富和快节奏的技术生活。

Manning出版社在此类古老书籍的插图中取材，基于两个世纪前丰富多样的地域生活来制作图书封面，借此颂扬计算机行业的创造力和首创精神。

目　录

第二部分　聚类

第 1 章

初识Mahout



本章内容

- ❑ Apache Mahout是什么？从哪里来？
- ❑ 现实中的推荐引擎、聚类和分类一览
- ❑ Mahout安装

大概你已经从书名中猜到了，本书主要讲解一个特殊的工具——Apache Mahout——在现实生活中的高效应用。它具备三个明显的特征。

首先，Mahout是一个来自Apache的、开源的*机器学习*（machine learning）软件库。它所实现的算法归属于机器学习或*集体智慧*（collective intelligence，也常常译为群体智慧或群体智能）这个广阔的领域。这意味着有许多事情可做，但对于此时此刻的Mahout，它主要关注于推荐引擎（协同过滤）、聚类和分类。

其次，Mahout是*可扩展的*。它旨在当所处理的数据规模远大于单机处理能力时成为一种可选的机器学习工具。在当前的Mahout系统中，这些可扩展的机器学习实现都是用Java来写的，而且有些部分是建立在Apache的Hadoop分布式计算项目之上的。

最后，它是一个*Java软件库*，并不提供用户接口、预装服务器（prepackaged server）或安装程序（installer）。它打算为开发者提供一个可用可改的工具框架。

出于阶段安排的需要，本章将通过一些常见的真实案例简要介绍一下推荐引擎、聚类和分类这几种机器学习手法，而Mahout通过它们帮助你处理数据。

若要在阅读本书时做到对Mahout随学随用，还要做一些必要的系统搭建与安装工作。

1.1 Mahout 的故事

首先来了解Mahout的背景知识。你可能还搞不清Mahout该如何发音：就是其通常的英语发音（[mə'haʊt]），它和trout押韵。它来自北印度语，意为驱象人，若要解释它，还有个小故事。

Mahout是2008年作为Apache Lucene的子项目出现的。Lucene项目推出了一个同名的著名开源搜索引擎，并给出了搜索、文本挖掘（text mining）和信息检索技术的先进实现方法。在计算机科学领域，这些术语和机器学习技术中的概念很相近，比如聚类（clustering），并在某种程度

上与分类（classification）相近。这样一来，某些Lucene贡献者的工作更多落入机器学习领域，从而逐渐脱离出来形成了独立的子项目。之后不久，Mahout吸纳了开源的协同过滤项目Taste。



图1-1给出了Mahout在ASF（Apache Software Foundation，Apache软件基金会）中的部分传承关系。到2010年4月，Mahout已经成为了一个独立的顶级Apache项目，并发布了一个全新的驱象人徽标。

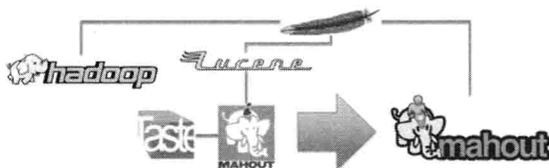


图1-1　Apache Mahout及其在ASF中的相关项目

Mahout所做的大量工作不仅体现在以高效和可扩展的方式实现这些经典算法，而且将部分算法进行转换使其可以在Hadoop上处理大规模的问题。Hadoop的吉祥物是一头象，Mahout项目的名字便由此而来！

从Mahout孵化出了许多技术和算法，其中有许多仍在开发或实验阶段（https://cwiki.apache.org/confluence/display/MAHOUT/Algorithms）。在该项目的早期，有3个明确的核心主题：推荐引擎（协同过滤）、聚类和分类。虽然它们绝非Mahout的全部，但在本书写作时，它们是最突出和最成熟的主题，也因此成为了本书的焦点。

也许你在阅读本书时已经了解了这三种技术的魅力，但为了不漏掉什么，请继续读下去。

1.2　Mahout 的机器学习主题

虽然Mahout项目在理论上可以实现所有类型的机器学习技术，但实际上当前它仅关注机器学习的三个主要领域，即推荐引擎（协同过滤）、聚类和分类。

1.2.1　推荐引擎

在目前采用的机器学习技术中，推荐引擎是最容易一眼就被认出来的。服务商或网站会根据你过去的行为向你推荐书籍、电影或文章。它们会推测你的品味与爱好，并找到某些你可能感兴趣的物品。

- 在部署了推荐系统的电子商务网站中，亚马逊大概是最有名的。亚马逊基于交易行为和网站记录为你推荐你可能感兴趣的书籍和其他物品（见图1-2）。
- 与之类似，Netflix为用户推荐其可能感兴趣的DVD，为了鼓励研究者改善其推荐质量，它给出了一份1 000 000美元的奖金，这使它颇具盛名。
- 像Líbímseti这样的约会网站（稍后讨论）还能把一个人推荐给另一个人。

- 而Facebook这样的社交网络则利用推荐技术为你找到最可能尚未关联的朋友。

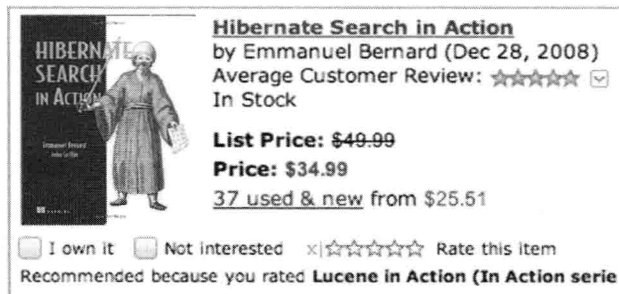


图1-2 亚马逊的推荐结果。基于该用户的交易历史和同类顾客的一些行为，亚马逊认为该用户会对这个推荐结果感兴趣。它甚至可以列出这次推荐的部分依据，即该用户已购或喜欢的类似物品

正如亚马逊等网站所展现的，推荐系统通过提供绝佳的交叉销售机会，从而产生实在的商业价值。某公司的报告显示，向用户推荐的产品能够使销售额增长8%~12%。[①]

1.2.2 聚类

聚类（clustering）的概念没有那么浅显易懂，但它也是在同样的应用场景下出现的。顾名思义，聚类技术试图将大量的事物组合为拥有类似属性的簇（cluster），借以在一些规模较大或难于理解的数据集上发现层次结构和顺序，以揭示一些有用的模式或让数据集更易于理解。

- Google News使用聚类技术通过标题把新闻文章进行分组，从而按照逻辑线索来显示新闻，而非给出所有文章的原始列表。如图1-3所示。
- 出于类似的原因，像Clusty这样的搜索引擎也将其查询结果进行分组。
- 聚类技术可以根据如收入、居住地和购买习惯等属性，将消费者分为许多段（簇）。

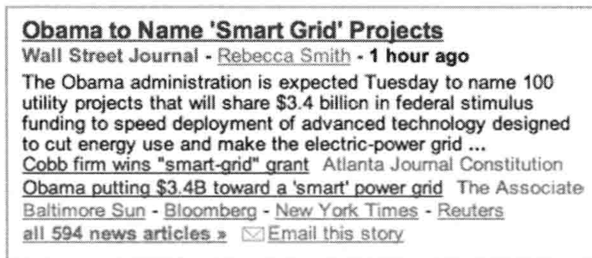


图1-3 Google News分组出的一段新闻样本。展现了一个代表性事件的详细片段，并且呈现了该主题下同一个集群内其他几个类似事件的链接。你也可以得到在该主题下聚类在一起的所有事件的链接

① 实用电子商务，“10 Questions on Product Recommendations”（产品推荐十问），http://mng.bz/b6A5。

聚类可以帮你在一大堆东西中找到脉络，甚至层次关系，否则理解这些数据将非常困难。利用这种技术，企业可以发现用户中潜在的群体，可以合理地组织大量文档，还可以根据日志来发现用户使用网站的常见模式。

1.2.3 分类

分类技术决定了一个事物多大程度上从属于某些类别或类型，或者多大程度上具有或不具有某些属性。与聚类一样，分类也无处不在，但是更多隐身于幕后。通常这些系统会考察类别中的大量实例，来学习推导出分类的规则。这种通常的方法有许多应用。

- 雅虎邮箱基于用户以前对正常或垃圾邮件的报告，以及电子邮件自身的特征，来判别到来的消息是否为垃圾邮件。几个被归类为垃圾邮件的消息如图1-4所示。
- 谷歌的Picasa和其他照片管理应用可以判断出一张照片中是否包含了人脸。
- OCR（Optical Character Recognition，光学字符识别）软件将一个个小块中的扫描文本分类成不同的字符。
- 据称iTunes中苹果公司的Genius特性使用分类来处理歌曲，为用户生成可能的播放列表。

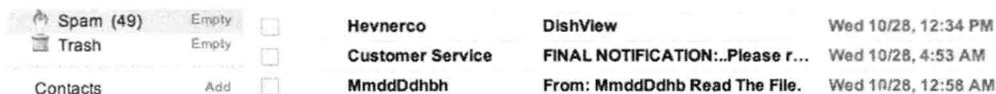


图1-4 由雅虎邮件检测出的垃圾消息。基于来自用户的垃圾邮件报告，结合其他分析，系统就能习得一些通常可用于确定垃圾邮件的属性。例如，提到“Viagra”的消息通常为垃圾邮件，故意错拼为“vlagra”的消息也一样。这些词项（term）[①]的出现就是垃圾邮件过滤器可以习得的一个属性

分类有助于判断一个新的输入或新的事物是否与以前观察到的模式相匹配，它通常还被用于遴选异常的行为或模式，来检测可疑的网络活动或欺骗行为。它还可用于“察觉”某个用户的消息是否存在失望或满意情绪。

如果输入数据的质量好，这些技术都可以完美地处理大量数据。但有时不仅需要处理大量的输入，还必须快速生成结果，这就使可扩展性（scalability）成为一个主要问题。并且，如前所述，Mahout存在的一个重要原因是能够为这些技术提供实现手段，从而使之向上扩展到处理庞大的输入数据。

1.3 利用 Mahout 和 Hadoop 处理大规模数据

规模问题在机器学习算法中有什么现实意义？让我们考虑你可能需要部署Mahout来解决的

① 词项是信息检索（information retrieval）领域的标准术语，意指用于表示查询或文档的特征，实际中文本常用单词（word）来表示词项，但是词项不一定就是单词。严格地说，词项、词条（token）和单词都不完全一样。本书并没严格区分。——译者注

几个问题的大小。

据粗略估计，Picasa三年前就拥有了5亿张照片。[①]这意味着每天有百万级的新照片需要处理。一张照片的分析本身不是一个大问题，即使重复几百万次也不算什么。但是在学习阶段可能需要同时获取数十亿张照片中的信息，而这种规模的计算是无法用单机实现的。

据报道，Google News每天都会处理大约350万篇新的新闻文章。虽然它的绝对词项数量看似不大，但试想一下，为了及时提供这些文章，它们连同其他近期的文章必须在几分钟的时间内完成聚类。

Netflix为Netflix Prize公布的评分数据子集中包含了1亿个评分。因为这仅仅是针对竞赛而公布的数据，据推测Netflix为形成推荐结果所需处理的数据总量与之相比还要大出许多倍。

机器学习技术必须部署在诸如此类的应用场景中，通常输入数据量都非常庞大，以至于无法在一台计算机上完全处理，即使这台计算机非常强大。如果没有Mahout这类的实现手段，这将是一项无法完成的任务。这就是Mahout将可扩展性视为重中之重的道理，以及本书将焦点放在有效处理大数据集上的原因，这一点与其他书有所不同。

将复杂的机器学习技术应用于解决大规模的问题，目前仅为大型的高新技术公司所考虑。但是，今天的计算能力与以往相比，已廉价许多，且可以借助于Apache Hadoop这种开源框架更轻松地获取。Mahout通过提供构筑在Hadoop平台上的、能够解决大规模问题的高质量的开源实现以期完成这块拼图，并可为所有技术团体所用。

Mahout中的有些部分利用了Hadoop，其中包含一个流行的MapReduce分布式计算框架。MapReduce被谷歌在公司内部得到广泛使用（http://labs.google.com/papers/mapreduce.html），而Hadoop是它的一个基于Java的开源实现。MapReduce是一个编程范式，初看起来奇怪，或者说简单得让人很难相信其强大性。MapReduce范式适用于解决输入为一组“键–值对”的问题，map函数将这些键值对转换为另一组中间键值对，reduce函数按某种方式将每个中间键所对应的全部值进行合并，以产生输出。实际上，许多问题可以归结为MapReduce问题，或它们的级联。这个范式还相当易于并行化：所有处理都是独立的，因此可以分布到许多机器上。这里不再赘述MapReduce，建议读者参考一些入门教程来了解它，如Hadoop所提供的http://hadoop.apache.org/mapreduce/docs/current/mapred_tutorial.html。

Hadoop实现了MapReduce范式，即便MapReduce听上去如此简单，这仍然称得上是一大进步。它负责管理输入数据、中间键值对以及输出数据的存储；这些数据可能会非常庞大，并且必须可被许多工作节点访问，而不仅仅存放在某个节点上。Hadoop还负责工作节点之间的数据分区和传输，以及各个机器的故障监测与恢复。理解其背后的工作原理，可以帮你准备好应对使用Hadoop可能会面对的复杂情况。Hadoop不仅仅是一个可在工程中添加的库。它有几个组件，每个都带有许多库，还有（几个）独立的服务进程，可在多台机器上运行。基于Hadoop的操作过程并不简单，但是投资一个可扩展、分布式的实现，可以在以后获得回报：你的数据可能会很快增长到很大的

① *Google Blogoscoped*，“Overall Number of Picasa Photos”（2007年3月12日），参见http://blogoscoped.com/archive/2007-03-12-n67.html。

规模，而这种可扩展的实现让你的应用不会落伍。

在第6章，本书将尝试克服这种复杂性，让你可以很快地在Hadoop上运行程序，之后你可以探索或研究关于集群操作和框架调优的细节。鉴于这种需要大量计算能力的复杂框架正变得越来越普遍，云计算提供商开始提供Hadoop相关的服务就不足为奇了。例如，亚马逊提供了一种管理Hadoop集群的服务Elastic MapReduce（http://aws.amazon.com/elasticmapreduce/），该服务提供了强大的计算能力，并使我们可通过一个友好的接口在Hadoop上操作和监控大规模作业，而这原本是一个非常复杂的任务。

1.4 安装 Mahout

之后的章节将会出现一些代码，你需要先配备一些工具才能随意使用这些代码。我们假设你对Java开发环境已经很熟悉了。

Mahout及其相关框架是基于Java实现的，因此具有平台独立性，你能够在任何一个可运行较新版JVM的平台上使用它。不过，我们有时仍然需要针对平台之间的差异性给出示例和解释。特别是在Windows shell上，其命令行命令与FreeBSD tcsh shell的不同。我们会使用bash中可用的命令和语法，它是大多数类Unix平台所采用的shell。默认情况下，大多数Linux发布包、Mac OS X、许多Unix变种和Cygwin（Windows上一种流行的类Unix环境）都使用它。打算使用Windows shell的用户对此很可能不习惯。不过，这些用户仍可以简单地使用本书所提供的代码清单，把命令翻译为在bash shell中可用的形式。

1.4.1 Java和IDE

如果做过Java开发，你的个人电脑上很可能已经安装了Java环境。注意，Mahout需要Java 6的支持。如果你不确定使用了哪个Java版本，可以打开一个终端并输入`java-version`查看。如果显示的版本低于1.6，你仍需要安装Java 6。

Windows 和 Linux 用户可以在 Oracle 找到 Java 6 的 JVM，网址为 http://www.oracle.com/technetwork/java/。苹果为Mac OS X 10.5和10.6提供了Java 6的JVM。在Mac OS X，如果显示所用版本不是Java 6，可以在/Applications/Utilities文件夹下打开Java Preferences应用。这里允许你将Java 6设为默认选项。

借助IDE（集成开发环境），大多数人可以轻松地编辑、编译和运行本书的示例；我们强烈推荐你使用IDE。Eclipse（http://www.eclipse.org）是最流行的免费Java IDE。本书不会涉及Eclipse的安装和配置，但继续阅读本书之前，你最好花点时间熟悉它。NetBeans（http://netbeans.org/）也是一个流行的免费IDE。另一个强大而流行的IDE是IntelliJ IDEA（http://www.jetbrains.com/idea/index.html），目前可获得免费的社区版本。

举一个使用IDE的例子，IDEA可以从现有的Maven模型中创建一个新的项目；如果你在创建项目时指定了Mahout源码的根目录，它会将整个项目按组织好的方式进行自动配置和展示。因此，我们可以将本书中所有的源代码放入examples/src/main/java/源码根目录中，并在IDEA中一键式运

行——依赖关系和编译的细节都会得到自动管理。这比手动编译和运行代码要容易得多。

注意 如果测试程序使用输入数据中的一个文件，它通常应该在与数据文件相同的目录中运行。查看你所用IDE的手册，了解如何为每个示例配置一个工作目录。

1.4.2 安装Maven

如同许多Apache的项目一样，Mahout利用Maven（http://maven.apache.org）来构建和发布项目。Maven是一个命令行工具，它管理依赖关系、编译代码、形成软件包、生成文档并发布正式版本。虽然它表面上类似于同样流行的工具Ant，实际却并不与之相同。Ant是一个灵活的低级脚本语言，而Maven是一个更重视依赖关系和发布管理的高级工具。鉴于Mahout使用了Maven，你最好把它安装好。

Mac OS X的用户会很高兴地发现Maven已经安装好了。如果没有，可以安装苹果的Developer Tools。在命令行输入`mvn --verison`。如果你成功地看到版本号，且版本大于或等于2.2，你就可以使用它了。否则，你需要在本地安装Maven。

用户若使用一个带有适当包管理系统的Linux发行版，便可以很快获得一个Maven的当前版本；否则就需要按照标准的流程进行安装，即下载一个二进制发行版，在一个类似/usr/local/maven的公共目录中解压，再编辑bash的配置文件~/.bashrc并添加一行，如`export PATH=/usr/local/maven/bin:$PATH`。它确保你随时可以使用mvn命令。

如果你正在使用Eclipse或IntelliJ这样的IDE环境，Maven已经被集成在其中了。参考其文档可以了解如何打开Maven集成的功能。这会大大简化Mahout在IDE中的使用，因为IDE可以使用一个项目中的Maven配置文件（pom.xml），来即刻配置并导入这个项目。

注意 对于Eclipse，你需要安装m2eclipse插件（http://www.eclipse.org/m2e/）。对于NetBeans，自6.7版之后就已经支持了Maven；而对于以前的版本，你需要额外安装一个插件。

1.4.3 安装Mahout

Mahout仍在不断发展，本书使用的是Mahout的0.5发布版。在https://cwiki.apache.org/confluence/display/MAHOUT/Downloads上可以找到下载这个发布版及其他版本的提示；你可以在计算机上找一个方便的地方将源码的压缩包解压。

因为Mahout的变更很频繁，定期会加入bug修复和一些改进，也许使用0.5的后续版本会更好（甚至可以用Subversion上仍未发布的最新代码，参见https://cwiki.apache.org/confluence/display/MAHOUT/Version+Control）。后续的发布包可以向后兼容地运行本书所提供的示例。

一旦你获得了源码，无论是从Subversion还是从发布包获得，都可以在IDE中为Mahout创建

一个新项目。IDE各不相同，参考其文档可以掌握它们在创建项目中的特殊用法。最简单的办法是使用IDE所集成的Maven，从项目源代码根目录中的pom.xml文件导入Maven项目。

一旦完成了这些步骤，你就可以很轻松地在这个项目中创建一个新的源代码目录，用来存放后续章节中所介绍的示例代码。正确地配置这个项目，以便可以顺利地编译并运行这些代码，这样你就无须付出额外的努力。

这些示例的源代码可以从Manning的网站（http://www.manning.com/MahoutinAction/）或GitHub（https://github.com/tdunning/MiA）上获得。你可以根据源码所提供的指导来建立你的工作环境。

1.4.4　安装Hadoop

你需要在本地安装一个Hadoop，来完成本书稍后所涉及的操作。你不必用一个集群来运行Hadoop。安装Hadoop虽不困难，但却有些烦琐。这里并不重复这个过程，我们将指导你从Hadoop网站http://hadoop.apache.org/common/releases.html获取一个0.20.2版的Hadoop副本，并遵照单节点安装文档（http://hadoop.apache.org/common/docs/current/single_node_setup.html）来安装一个伪分布模式的Hadoop。

1.5　小结

Mahout来自Apache，它是一个“年轻”、开源、可扩展的机器学习库，而本书将指引你在Mahout上使用机器学习技术解决实际问题。特别地，你会很快了解推荐引擎、聚类和分类。如果你是一个熟知机器学习理论的研究者，正在寻找一个实用的how-to指南，或者是一个希望快速掌握从业者宝贵经验的开发者，那么这本书正是为你而写。

这些技术已经不再只是理论。我们已经知道在现实世界中有许多广为人知的机器学习案例，它们采用了推荐引擎、聚类和分类：电子商务、电子邮件、视频网站、照片网站以及更多。这些技术已经被用于解决实际问题，甚至为企业创造价值——现在它们都可以借助Mahout来实现。

我们已经发现这些技术有时会涉及大量的数据——可扩展性是这个领域中一个独特而永恒的话题。初步审视MapReduce和Hadoop，我们了解到它们是如何承载了Mahout所提供的可扩展性。

因为本书注重实际操作，所以我们让你一上来就准备好去使用Mahout。现在，你应该已经安装了Mahout工作所需的工具，并准备开始行动了。因为本书旨在实战，让我们现在就结束开篇，来看看Mahout的实际代码。请看后续篇章！

Part 1 第一部分

推　荐

本书第一部分涵盖第2章至第6章，探讨Apache Mahout机器学习实现的三大支柱之一：协同过滤（collaborative filtering）和推荐（recommendation）。通过这些技术，你能够了解一个人的品味，并自动找到新内容来投其所好。本部分仍为后续章节的铺垫，后续章节将高度依赖于Apache Hadoop的分布式计算框架。我们先通过简单的Java程序来了解Apache Mahout的机器学习，再使用Hadoop来实现它。

第2章介绍由Mahout实现的推荐引擎（recommender engine），并在一个可运行的示例中评价性能。第3章讨论Mahout中推荐程序（recommender）的高效数据表示。第4章分类说明Mahout中推荐引擎的各种实现及其不同的属性特征。

第5章给出一个实例，其数据来自一个约会网站，由此讨论如何采用Mahout中的方法来处理真实数据，从而形成一个可供生产环境使用的推荐程序。最终，第6章会初步在Apache Hadoop上使用Mahout，以实现一个大型的分布式推荐引擎。

第2章 推荐系统

本章内容

- ❑ Mahout中的推荐系统
- ❑ 推荐系统实战初探
- ❑ 评估推荐引擎的精度和质量
- ❑ 评估基于实际数据集GroupLens的推荐程序

我们每天都对事物形成观点：喜欢、不喜欢，甚或不关心。这都是无意识中发生的。当你在广播中听到一首歌时，你可能因为它动听而注意它，也可能因为它难听而注意它，也有可能压根儿就没有注意到它。同样的情形还适用于T恤衫、色拉、发型、滑雪场、容貌和电视节目。

人们的嗜好各异，却有规律可循。人们倾向于喜欢那些与其爱好相似的东西。由于Sean喜欢吃火腿-莴苣-番茄三明治，你就可以猜测他可能会喜欢总会三明治（club sandwich），因为它们基本上是一样的，只是后者使用了火鸡肉。而且，人们容易爱上类似人群所喜欢的东西。

这些模式可用于预测人们的好恶。推荐就是通过对嗜好的这些模式进行预测，借以发现你尚未知晓，却合乎心意的新事物。

在更深入地介绍推荐思想之后，本章将帮助你体验Mahout的一段代码，用以运行一个简单的推荐引擎并了解其执行效果，从而让你直观感受一下Mahout是如何实现推荐的。

2.1 推荐的定义

你从书架上拿起这本书是有原因的。也许它恰好放在对你有用的其他书籍的旁边，而你明白之所以书店会把它放在那里，是因为喜欢那些书的人很可能也会喜欢这本书。或许它恰好放在你同事的书架上，而你们在机器学习方面志趣相投，也有可能是他们直接向你推荐了本书。

这些策略虽然各不相同，但在发掘新鲜事物上都是有效的：要找到你可能喜欢的物品，你可以观察与你志趣相投的人喜欢些什么。另一方面，通过观察其他人的明显偏好，你可以弄清楚哪些东西和你已然喜欢的物品相似。实际上，它们是推荐引擎算法中应用最广的两大类：基于用户（user-based）和基于物品（item-based）的推荐程序，它们均在Mahout中得到了充分展现。

严格说来，上述场景均为协同过滤的范例——仅仅通过了解用户与物品之间的关系进行推荐。这些技术无须了解物品自身的属性。从某种意义上讲，这是一个优点。该推荐框架并不关心物品是否为书籍、主题公园、鲜花或其他人，因为根本不会导入它们的属性。

其他一些方法则立足于物品的属性，通常称为基于内容（content-based）的推荐技术。例如，如果有朋友向你推荐本书，原因是它是Manning出版的，而且他也喜欢Manning出版的其他书，那么这个朋友所做的就是类似于基于内容的推荐。它给你的建议是基于书的属性，即基于“出版商”作出的。

基于内容的推荐技术没有什么问题，相反，它们很有用。但是，它们必须与特定领域相结合，而难以规整为一个框架。为了构造一个有效的基于内容的图书推荐程序，人们不得不确定图书的哪种属性（页数、作者、出版商、颜色、字体）是有意义的，以及有多大意义。这些知识无法转换以用于其他领域，比如这种推荐图书的方法对于推荐比萨配料毫无用处。

因此，Mahout对基于内容的推荐所言甚少。这些思想能够融入并构建在Mahout之上；故而，Mahout在技术上可称为一种协同过滤框架。第5章会给出一个示例，指导你为约会网站创建一个推荐程序。

但在现阶段，我们先生成一些简单的输入并据此找出推荐结果，来体验一下Mahout中的协同过滤。

2.2　运行第一个推荐引擎

Mahout包含一个推荐引擎，其中有几种类型实际来自于传统的基于用户和基于物品的推荐程序。它也包含了其他几种算法实现，但是现在我们先看一个简单的基于用户的推荐程序。

2.2.1　创建输入

为了探究Mahout中的推荐，最好从一个简单的例子开始。

推荐程序需要有输入——构成推荐的基础数据。在Mahout的语言中，数据是以偏好（preference）的形式来表达的。因为最常见的推荐引擎总是把项目推荐给用户，所以谈论偏好最简便的方法是建立从用户到物品的关联，尽管如前所述，这些用户和物品是可以任意指定的。一个偏好包含一个用户ID、一个物品ID，通常还有一个表达用户对物品的偏爱程度的数值。实际上，Mahout中的ID通常也为数字——整数。偏好值（preference value）可任意设定，只需保证更大的值代表更强的正向偏好。例如，这些值可能按从1到5来定级，其中1表示用户非常不喜欢该物品，而5表示物品是用户的至爱。

创建一个包含关于用户数据的文本文件，巧妙地从1到5为用户命名，从101到107为他们喜欢的7本书命名。在现实世界，这些数据可能是来自某公司数据库的顾客ID和产品ID；Mahout并不要求用户和物品必须按照数字命名。我们用一种简单的以逗号分隔值的格式把这些数据写入文件。

复制下面的示例到一个文件中并将之存为intro.csv。

代码清单2-1　推荐程序的输入文件intro.csv

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

用户ID、物品ID、偏好值

用户1对物品102的偏好值为3.0

一番研究之后，倾向就明显了。用户1和5似乎有相似的喜好。他们都最喜欢101这本书，其次喜欢102，再次喜欢103。同样，对于用户1和4，他们似乎都喜欢101和103（但用户4对102的关系不明）。另一方面，用户1和2的喜好基本上是对立的：用户1喜欢101，而用户2对其不感兴趣；用户1喜欢103，而用户2则恰恰相反。用户1和3的喜好迥异——他们仅同时喜欢101。图2-1显示了用户和物品之间正面和负面的关系。

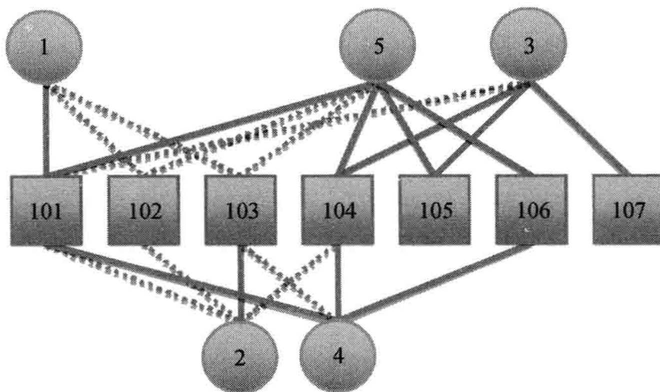


图2-1　用户1到5和物品101到107的关系。虚线表示看似负面的关系，即用户似乎不太喜欢这个物品，但也表达了对物品的态度

2.2.2 创建一个推荐程序

那么，可以为用户1推荐什么书呢？不是101、102和103，因为用户1显然已经知道这些书了，而推荐是用来发现新事物的。直观上看，既然用户4和5与用户1类似，那么把用户4或5喜欢的东西推荐给用户1是个好主意。这样一来，可能的推荐结果就是书104、105和106。总体上看，104似乎最有可能，因为物品104对应的偏好值是4.5和4.0。

现在，运行如下代码。

代码清单2-2 一个简单的基于用户的Mahout推荐程序

```
class RecommenderIntro {

  public static void main(String[] args) throws Exception {

    DataModel model =
      new FileDataModel (new File("intro.csv"));          <- 装载数据文件

    UserSimilarity similarity =
      new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood (2, similarity, model);

    Recommender recommender = new GenericUserBasedRecommender (
        model, neighborhood, similarity);                  <- 生成推荐引擎

    List<RecommendedItem> recommendations =
        recommender.recommend(1, 1);                         <- 为用户1推荐物品1

    for (RecommendedItem recommendation : recommendations) {
      System.out.println(recommendation);
    }

  }

}
```

图2-2形象地表示了这些基础组件之间的关系。并非所有基于Mahout的推荐程序都是如此，有些会采用不同的组件、不同的关系。但这个例子先让我们对此有一些感觉。

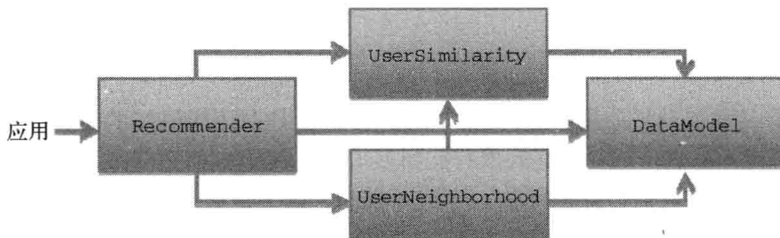


图2-2 Mahout基于用户推荐程序中组件间关系的简单示意图

在接下来的两章中，我们将详细地逐一讨论这些组件，但现在先对每个组件的角色做一个概览。DataModel实现存储并为计算提供其所需的所有偏好、用户和物品数据。UserSimilarity

实现给出两个用户之间的相似度，可从多种可能度量或计算中选用一种来作为依据。UserNeighborhood实现明确了与给定用户最相似的一组用户。最后，Recommender实现合并所有这些组件为用户推荐物品。

2.2.3　分析输出

当运行代码清单2-2中的代码时，你的终端或IDE（集成开发环境）会输出如下结果：

```
RecommendedItem [item:104, value:4.257081]
```

该请求寻找一个最优的推荐结果，并最终找到了一个。推荐引擎把书104推荐给用户1。而且推荐引擎之所以这样做，是因为它估计出用户1对书104的偏好值约为4.3，而这在所有适合推荐的物品中是最高的。

推荐结果还不错。没有出现书107；它虽然也在可被推荐之列，但它仅和一个嗜好不同的用户相关。推荐引擎选择104而没选106也是合理的，因为能看到104的总体评分略高。此外，输出还包含了一个合理的估计，即用户1有多喜欢物品104——大致在用户4和5所表达的偏好值4.0和4.5之间。

从数据中不能一眼看出正确答案，但是推荐引擎找到了它的踪迹，并返回了一个合理的答案。这个简单的程序找到了一个不易发现的有用结果，如果你为此感到欢欣鼓舞，那就表示机器学习的世界很适合你。

对于干净的小数据集，生成推荐结果就像前面的示例一样简单。但现实中，数据集往往非常庞大，而且其中很多信息没有价值。例如，假设一个受欢迎的新闻网站要为读者推荐新闻文章。可以根据文章点击率推断出偏好，但也可能会产生很多假的偏好——或许读者点击了并不喜欢的文章，或错误地点击了一个故事。或许很多点击发生在未登录状态下，因而不能与某个用户进行对应。再试想一下数据集的大小——也许每个月的点击量有几十亿次。

要在该数据之上快速生成准确的推荐结果并不简单。后面的案例研究中，我们将使用Mahout提供的工具来解决一组这样的问题。它们会为你呈现标准的方法会如何导致糟糕的推荐结果，或是耗费大量的内存和CPU时间，同时展示如何配置和定制Mahout来提高性能。

2.3　评估一个推荐程序

推荐引擎是一种工具，一种解答问题的手段。“什么是对用户最好的推荐？”在探寻其答案之前，最好先深究一下这个问题。好的推荐需要多准确？用户如何获知推荐程序正在输出最佳结果？本章后续部分将转而探讨如何评估一个推荐程序，因为这会有助于审视特定的推荐系统。

最佳的推荐程序应该就像是一个“巫师”，它能够在你行动之前设法准确地获知你喜欢的每一种可能的物品，而且这些物品是你尚未见过或没有对其表达过任何喜好意见的。能够准确预测你所有喜好和行为的推荐程序还应按你未来的喜好把物品进行排队。最优的潜在推荐结果应该就是这样。

而实际上，大多数推荐引擎仅会试图给出某些或其他所有物品的估计评分。由此，一种评估推荐程序推荐结果的方法是评估其估计偏好值的质量——即评估所估计的偏好在多大程度上与实际偏好相匹配。

2.3.1 训练数据与评分

但是，没有现成的实际偏好值可用。没有人能确切地知道你将来有多喜欢某些新东西（包括你在内）。在推荐引擎中，这可以通过提取一小段真实数据作为测试数据来*仿真*。这些用于测试的偏好不会作为训练数据导入到被评估的推荐引擎。相反，推荐程序需要为这些缺失的测试数据估计出偏好值，然后估计结果用于与真实值进行对照。

进而，我们可以非常简单地为推荐程序做一种评分。例如，可以计算出在估计和实际偏好之间的平均差值。在这种评分中，值*越低越好*，因为值越低意味着估计值与实际偏好值的差别越小。评分为0.0意味着完美的估计，即在估计值和实际偏好值之间根本没有差别。

有时会使用差值的均方根：计算出实际偏好值和估计值之间的差值之后，先进行平方再求其均值的平方根。见表2-1。值同样是越低越好。

表2-1 平均差值与均方根的计算说明

	物品1	物品2	物品3
真实值	3.0	5.0	4.0
估计值	3.5	2.0	5.0
差值	0.5	3.0	1.0
平均差值	=(0.5+3.0+1.0)/3=1.5		
均方根	$=\sqrt{((0.5^2+3.0^2+1.0^2)/3)}=1.8484$		

表2-1显示了一组实际偏好和估计之间的差值，以及如何将它们转换为评分。均方根使得估计值的偏离显得更严重，正如这里的物品2，这在有时是需要的。例如，相比于偏离1颗星的估计值，偏离2颗星对推荐所造成的不良影响也许会超过2倍。鉴于简单地对差值求平均可能更直观和易于理解，后续的例子中都会采用这种方法。

2.3.2 运行RecommenderEvaluator

让我们重温示例程序，在简单的数据集上评估这个简易的推荐程序，如下列代码清单所示。

代码清单2-3 配置并评估一个推荐程序

```
RandomUtils.useTestSeed();                                          <- 生成可重复的结果
DataModel model = new FileDataModel (new File("intro.csv"));

RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator ();
```

```
RecommenderBuilder builder = new RecommenderBuilder() {        构建如代码清单2-2所
  @Override                                                    示的推荐程序
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood (2, similarity, model);
    return
      new GenericUserBasedRecommender (model, neighborhood, similarity);
  }
};
double score = evaluator.evaluate(                  训练70%的数据，测试30%
    builder, null, model, 0.7, 1.0);
System.out.println(score);
```

大多数行为发生在evaluate()中：RecommenderEvaluator将数据分为训练集和测试集，构建一个新训练的DataModel与Recommender用于测试，并将估计的偏好值与实际测试数据进行比较。

注意传递给evaluate()的参数中没有Recommender。这是因为在该方法中，Recommender是由新训练的DataModel来构建的。该方法的调用者必须提供一个对象——RecommenderBuilder，可以使用DataModel构建出Recommender。这里，该方法所用的是和本章之前所述相同的实现。

2.3.3　评估结果

代码清单2-3中的程序输出评价的结果：一个显示Recommender表现如何的分数。在这个例子中，你只会看到1.0这一个值。即使evaluator在选择测试数据时引入许多随机量，结果仍是相同的，因为对RandomUtils.useTestSeed()的调用会强制每次选择相同的随机值。这仅仅是为了获得可重复的结果，而被用在这样的示例或单元测试中。请不要在实际代码中这样用！

这个分值的意义取决于所采取的实现方法，这里是AverageAbsoluteDifference-RecommenderEvaluator。在该实现中分值为1.0，这意味着平均而言推荐程序所给出的估计值与实际值的偏差为1.0。

在从1至5的区间中，1.0这个值并不大，但我们这里只采用了非常少的数据。你来执行时所获得的结果也许会不同，因为对数据集的分片是随机的，而且程序每次运行所用的训练集和测试集也可能不一样。

这个技术可以应用于任何Recommender和DataModel。如要使用均方根来评分，可以用RMSRecommenderEvaluator取代AverageAbsoluteDifferenceRecommenderEvaluator。

你可以选择不向evaluate()传递null（空）参数，而是传递DataModelBuilder的一个实例（instance），它可以用于控制如何从训练数据中生成DataModel。通常默认地传空参数就够了，除非你使用了一个特殊的DataModel实现——一个你希望插入到评估过程中的DataModelBuilder。

最后传递给evaluate()的参数1.0是用来控制总共使用多少输入数据的。这里，它是指100%的数据。这个参数可用于仅通过庞大数据集中的很小一部分数据，来生成一个精度较低但

更快的评估。例如，0.1代表使用10%的数据，而90%的数据被忽略。当你希望快速测试Recommender中一些小的更改时，这个参数会很有用。

2.4 评估查准率与查全率

我们还应该更全面地看待推荐问题：通过估计偏好值来生成推荐结果并非绝对必要。给出一个从优到劣排列的推荐列表对于许多场景都够用了，而不必包含估计的偏好值。事实上，有时精确的列表顺序也不那么重要——有几个好的推荐结果就可以了。

从这种更普遍的视角，我们还可以运用经典的信息检索（information retrieval）度量标准来评估推荐程序：查准率（precision）和查全率（recall）。这些术语通常用在像搜索引擎这样的系统中，即从许多可能的搜索结果中返回一组最佳结果。

搜索引擎应避免在top结果中返回无关信息，而应竭力返回尽可能相关的结果。在一些对“相关”的定义中，查准率是指在top结果中相关结果的比例。“Precision at 10”（推荐10个结果时的查准率）是指这个比例来自对前10个top结果的判定。查全率是指所有相关结果包含在top结果中的比例。图2-3给出了它们的图示。

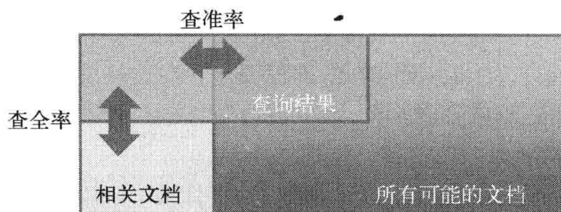


图2-3 在搜索结果中查准率和查全率的说明

这些术语很容易用在推荐程序中：查准率是top推荐中间有“好”结果的比例，而查全率是“好”结果出现在top推荐中的比例。下一节将定义何为“好”。

2.4.1 运行RecommenderIRStatsEvaluator

Mahout同样提供了一个相当简单的方法，为Recommender计算出这些值，如下面的代码清单所示。

代码清单2-4 查准率和查全率评估的配置与运行

```
RandomUtils.useTestSeed();
DataModel model = new FileDataModel (new File("intro.csv"));

RecommenderIRStatsEvaluator evaluator =
  new GenericRecommenderIRStatsEvaluator ();
```

```
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood (2, similarity, model);
    return
      new GenericUserBasedRecommender (model, neighborhood, similarity);
  }
};
IRStatistics stats = evaluator.evaluate(
    recommenderBuilder, null, model, null, 2,
    GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD,
    1.0);                                    <- 评估推荐2个结果时的查准率和查全率

System.out.println(stats.getPrecision());
System.out.println(stats.getRecall());
```

如果不调用RandomUtils.useTestSeed()，你会看到完全不同的结果，因为训练数据和测试数据是随机选择的，而且这里选用的数据集也非常小。但是加入这个调用之后，结果就应该为：

```
0.75
1.0
```

“Precision at 2”（推荐2个结果时的查准率）为0.75；平均有3/4的推荐结果是好的。“Recall at 2”（推荐2个结果时的查全率）为1.0；所有好的推荐都包含在这些推荐结果中。[①]

但是，到底什么才是好的推荐呢？框架要负责作出决定，而没有人给它一个定义。直观上看，在测试集中最受欢迎的物品为好的推荐，其他则不是。

代码清单2-5 在测试数据集中用户5的偏好值

```
5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

重新看一下该样本数据集中的用户5。我们把物品101、102和103的偏好值分离出来作为测试数据。它们的偏好值分别为4.0、3.0和2.0。当这些值不在训练数据集中时，推荐引擎应该先推荐101，再是102，最后是103，因为这是用户5对这些物品的偏好顺序。但是推荐103会不会是个好主意呢？它位于列表的末尾，用户5不会很喜欢它。而用户5对书102的喜好也只是一般而已。书101看起来不错，因为它的偏好值远远超过平均值。或许101是一个好的推荐，102和103也不错，但算不上是好的推荐。

但这是RecommenderEvaluator的思维方式。当没有明确的阈值可将推荐分出好坏时，框架会为每个用户取一个阈值，它等于该用户的平均偏好值μ，加上一个标准方差σ：[②]

---

① 查准率和查全率均为对所有用户的推荐分别评估后取的平均值。——译者注

② 注意这里讨论的是RecommenderIRStatsEvaluator中阈值的设定，RecommenderEvaluator直接将估计值与实际值相比较，故不需使用阈值。——译者注

$$阈值 = \mu + \sigma$$

即使你已经忘记了统计数据，也没关系。确定阈值所用的物品偏好值不是略高于平均值（μ），而是比平均值高出很多（σ）。在现实场景下，这意味着大约有16%的物品最受欢迎，它们可以被视为好的推荐并反馈给用户。该方法所用的其他参数和以前类似，它们在项目的Javadoc中有完整的文档说明。

2

2.4.2　查准率和查全率的问题

No. 2

在推荐程序中，查准率和查全率测试的有效性完全依赖于怎样定义“好的推荐”。在前一节中，阈值要么是特别指定的，要么是由框架定义的。阈值选择不当会损害到对推荐结果评分的有效性。

但是，这些测试还有一个更细节的问题。这里，它们必然是从那些用户已经表达过一些偏好的物品中挑选一组好的推荐结果。但是，最好的推荐结果并不一定在那些用户已知的物品中！

试想为一个用户运行这个测试，这个用户肯定喜欢小众的法国非主流电影*My Brother the Armoire*。平心而论，这是给用户的一个非常棒的推荐，但这个用户从来没有听说过这部电影。假如推荐程序推荐这部电影，会被认为是推荐错误；测试框架仅会从用户已有的偏好集合中选择好的推荐。

如果偏好是布尔型，不包含偏好值，那么事情就更复杂了。这时，甚至没有相对偏好的概念可用于选出包含好物品的数据子集。该测试可做的最好选择就是随机选择一些受欢迎的物品作为好的推荐。

这个测试仍然有些用处。用户偏好的物品可以很好地代表对用户的最佳推荐，不过它们绝非完美的选择。在布尔型偏好数据的案例中，只能做查准–查全测试（precision-recall test）。理解这个测试在该场景下的局限是必要的。

2.5　评估 GroupLens 数据集

有这些工具在手，我们不仅可以评估推荐引擎的速度，还可以评估其质量。虽然几章之后才会讨论有关大规模真实数据的示例，但现在我们已经可以快速评估一个小数据集的性能了。

2.5.1　提取推荐程序的输入

GroupLens（http://grouplens.org/）是一个研究项目，提供多个大小不同的数据集，每个都来自真实用户对电影的评分。它是几个可用的大规模真实数据集中的一个，本书稍后还会为你介绍更多的数据集。

在GroupLens网站上，找到并下载“100K data set”，当前其地址为http://www.grouplens.org/node/73。将下载的文件解压，在其中找到名为ua.base的文件。这是一个以制表符（tab）分隔的文件，包含用户ID、物品ID、评分（偏好值），以及一些附加信息。

这个文件的字段用制表符分隔，而不是逗号，结尾还包含一个额外的信息字段。它可用吗？是的，这个文件可用于`FileDataModel`。回到代码清单2-3的代码，创建一个`Recommender-Evaluator`，然后把ua.base的位置传递给它，而不再是传递一个小数据文件。再次运行。这次，

评估会花上几分钟，因为现在是基于100 000个偏好值，而不是少数几个。

最终，你会得到一个大约为0.9的值。这不算坏，但放在1到5的区间内，这个值偏离了将近1个点，看起来不算太好。对这类数据，也许我们正在使用的这个特定的Recommender实现并不是最优的？

2.5.2 体验其他推荐程序

让我们试着在这个数据集上运行一个slope-one推荐程序，这是一个还会在第4章中出现的简单算法。如下所示，用org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOne-Recommender替代RecommenderBuilder即可。

代码清单2-6　改变评估程序后运行SlopeOneRecommender

```
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
    return new SlopeOneRecommender(model);
  }
};
```

再次运行该评估。你会发现它快了很多，而且生成的评估结果大约为0.748。正在向正确的方向前进。

这并不是说slope-one总是更好或更快。每个算法都有其特征与属性，无法预知它们在给定数据集上的行为。例如，虽然slope-one在运行时会很快算出推荐结果，但在运行之前却需要大量的时间来预先算出其内在的数据结构。因此，我们最初介绍的基于用户的推荐程序也许在其他数据集上会更快和更准确。第4章会探讨每种算法的相对优势。

这种区别彰显了在真实数据上做测试与评估的重要性，以及使用Mahout如何相对消除了一些麻烦。

2.6 小结

本章中，我们介绍了推荐引擎的思想。我们选用一个简单的Mahout Recommender，为之创建了一些小规模的输入数据、运行了一个简单计算，并解释了其结果。

接着我们评估了推荐引擎输出结果的质量，这是在后续章节中需要经常使用的。本章涵盖对Recommender所估计偏好的精度评估，以及传统的查准率和查全率度量标准在推荐中的应用。最终，我们尝试评估一个来自GroupLens的真实数据集，并观察如何借由评估在现实场景中探索对推荐引擎的改进。

在我们继续详解推荐引擎之前，还有一个重要的事情要做，即了解Mahout中推荐程序的另一个基本概念：数据表示。我们将在下一章讨论它。

第3章

推荐数据的表示

本章内容

- ❑ Mahout如何表示推荐数据
- ❑ DataModel的实现和用法
- ❑ 无偏好值时的数据处理

推荐的质量很大程度上取决于数据的数量和质量。"种瓜得瓜，种豆得豆"，没有比用在这里更恰当的了。拥有高质量的数据当然是件好事，而且通常越多越好。

但是，推荐算法天生是数据密集型的，其计算涉及对大量信息的访问。因此，数据的数量和表示方式会很大程度上影响执行性能。智能地选择数据结构能够极大地改善性能，数据达到一定规模的时侯，这并非小事。

本章探讨Mahout在表示和访问推荐程序的相关数据时所用的关键类。你会更好地理解为什么Mahout采用这样的方式来表示用户和物品及其相关的偏好，以达到高效和可扩展性。本章还会详细解析在Mahout中用于访问数据的关键抽象：DataModel。

最后，让我们来看看当用户和物品的数据没有评分或偏好值时的情况，即所谓的布尔偏好（Boolean preference），这时就需要做特殊的处理。

第一节介绍推荐数据的基本单元：用户对物品的偏好（user-item preference）。

3.1 偏好数据的表示

推荐引擎的输入是偏好数据（preference data）：什么人喜欢什么物品以及喜欢的程度。这意味着该输入就是一个用户ID、物品ID和偏好值的元组集合——这自然是一个大数据集。有时，偏好值会被忽略。

3.1.1 Preference对象

Preference是最基本的抽象，表示单个用户ID、物品ID和偏好值。一个对象代表一个用户对一个物品的偏好。Preference是一个接口，你最有可能使用的实现是GenericPreference。例如，下面一行代码所生成的表示形式意味着用户123对于物品456的偏好值为3.0：

```
new GenericPreference(123, 456, 3.0f)
```

那么一组Preference该如何表示呢？如果你给出像Collection<Preference> 或者Preference[]这类答案，虽然看似合理，但对于大多数Mahout API而言通常都是错误的。聚合（collection）和数组（Array）在表示大量Preference对象时会变得相当低效。如果你从未见识过Java中一个Object的开销，你一定会被吓到！

一个GenericPreference包含20字节的有用数据：一个8字节的用户ID（Java long）、一个8字节的物品ID（long）和一个4字节的偏好值（float）。而该对象的存在所需要的开销令人吃惊：28字节！这个对象的表示形式包含一个8字节的对该对象的引用，以及由于Object开销和其他对齐问题所带来的另外20字节。于是GenericPreference对象仅由于引用的开销上就比实际多消耗了1.4倍的内存。

注意　实际的开销大小因JVM实现而不同；上述数据是针对苹果Mac OS X 10.6的64位Java 6虚拟机而言的。

该如何表示大量Preference对象呢？在推荐算法中，通常需要一个与某个用户或某个物品关联的所有偏好的聚合。在这种聚合里，所有Preference对象的用户ID或物品ID都是一样的，这似乎是冗余的。

3.1.2 PreferenceArray及其实现

看一下PreferenceArray，这是一个接口，它的实现表示一个偏好的聚合，具有类似数组的API。例如，GenericUserPreferenceArray表示的是与某个用户关联的所有偏好。其内部包含一个单一用户ID、一个物品ID数组，以及一个偏好值数组。在这个表示形式中，每个偏好的边界内存（marginal memory）仅需要12字节（一个数组有一个8字节的物品ID和一个4字节的偏好值）。与此对应，一个完整的Preference对象需要大约48字节。这种特殊的实现仅在内存上就节省了4倍空间，而且需要由垃圾回收器分配和检查的对象也少多了，因此性能也能获得一定的提升。比较图3-1和图3-2就能理解这种节省是如何达成的。

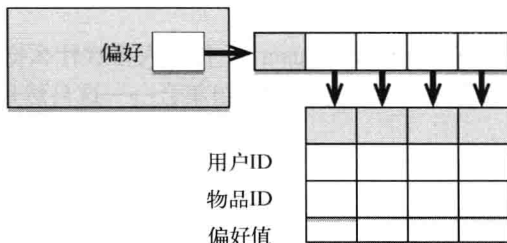


图3-1　一种基于Preference对象数组的相对低效的偏好表示形式。灰色区域大体表示Object的开销。白色区域为数据，包括Object的引用

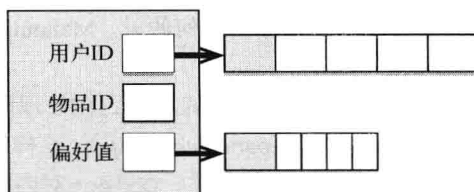


图3-2 基于GenericUserPreferenceArray的更高效的表现形式

下列代码显示了PreferenceArray典型的构建和访问方式。

代码清单3-1 设置PreferenceArray中的偏好值

```
PreferenceArray user1Prefs = new GenericUserPreferenceArray(2);

user1Prefs.setUserID(0, 1L);                  <-- 设置这些偏好的用户ID

user1Prefs.setItemID(0, 101L);
user1Prefs.setValue(0, 2.0f);

user1Prefs.setItemID(1, 102L);                | 表示这些偏好
user1Prefs.setValue(1, 3.0f);                         提取物品102的
                                                      Preference
Preference pref = user1Prefs.get(1);          <--
```

同样，存在一个称为GenericItemPreferenceArray的实现，它封装了所有与某一物品相关联的偏好，而不是关联到某个用户。它的用途与用法完全类似。

3.1.3 改善聚合的性能

你可能会想："太棒了！Mahout已经创造了一个Java对象的数组。"哦，先别急，因为还有惊喜。你还记得我们曾提到过规模的重要性吗？希望你已经明白使用这种技术将要面对的数据大得非同寻常，而这可能会带来出乎意料的后果。

PreferenceArray及其实现降低了对内存的需求，即便引入复杂性也是值得的。将内存需求砍掉3/4并不只是节省了几兆字节——在一定规模下这会节省出几十GB的内存容量。这也许就是你的现有硬件能否容纳下这些数据的区别。也许这意味着你是否需要花费许多钱来购买更多RAM，或者一个新的64位系统。这是一个看似很小，却很实在的节省。

3.1.4 FastByIDMap和FastIDSet

你一定不会吃惊，Mahout的推荐程序中大量使用了Map和Set这些典型的数据结构，但它们用的并不是通常的Java集合（collection）的实现，如TreeSet和HashMap。相反，通览全部的实现与API，你会找到FastMap、FastByIDMap和FastIDSet。它们类似于Map和Set，但做了特殊定制，仅为满足Mahout中推荐程序的需要。它们降低了对内存的占用，而不是去显著地改善性能。

不能把它们当做是对Java Collections框架的批评。相反，集合因为良好的设计，可以有效地

适用于很多场景。只是，它们无法对使用方式作出任何假设。Mahout的需求则更具针对性，从而可以对用途作出更强的设定。主要区别如下。

- 与HashMap类似，FastByIDMap是基于散列的。但它在处理散列冲突时使用的是线性探测（linear probing），而非分离链接（separate chaining）。这样便不必为每个条目（entry）都增加一个额外的Map.Entry对象；如前所述，Object对内存的消耗是惊人的。
- 在Mahout推荐程序中键（key）和成员（member）通常采用原始类型long，而非Object。使用long型的键可以节约内存并提升性能。
- Set实现的内部没有使用Map。
- FastByIDMap可以作为高速缓存，因为它有一个最大空间的概念；超过这个大小时，若要新加入条目则会把不常用的移走。

存储上的差异是非常明显的：FastIDSet平均每个成员需要大约14字节，而HashSet需要84字节。FastByIDMap每个条目需要大约28字节，而HashMap每个条目需要大约84字节。这说明当能够在用途上作出更强假设时，就有可能进行大幅的优化——这里主要是在内存需求上。考虑到推荐系统所处理的数据量，这些定制化的实现并非自卖自夸。

那么，这些精心设计的类被用在哪里了呢？

3.2　内存级 DataModel

在Mahout中使用DataModel这种抽象机制对推荐程序的输入数据进行封装，而DataModel的实现为各类推荐算法提供了对数据的高效访问。例如，DataModel可以提供输入数据中所有用户ID的计数或列表、提供与某个物品相关的所有偏好，或给出所有对一组物品ID表达过偏好的用户的个数。

本节仅关注一些要点；更多关于DataModel的内容参见在线的Javadoc文档（https://builds.apache.org/job/Mahout-Quality/javadoc/）。

3.2.1　GenericDataModel

内存级（in-memory）实现GenericDataModel是现有DataModel实现中最简单的。它适用于通过程序在内存中构造数据的表示形式，而不是基于来自外部的数据源，如文件或关系数据库。它简单地将偏好作为输入，采用FastByIDMap的形式，将用户ID映射到这些用户的数据所在的PreferenceArray上，如下所示。

代码清单3-2　利用GenericDataModel在程序中定义输入数据

```
FastByIDMap<PreferenceArray> preferences =
  new FastByIDMap<PreferenceArray>();
PreferenceArray prefsForUser1 = new GenericUserPreferenceArray(10);

prefsForUser1.setUserID(0, 1L);
prefsForUser1.setItemID(0, 101L);          增加10个偏好中的第1个
prefsForUser1.setValue(0, 3.0f);
```

```
prefsForUser1.setItemID(1, 102L);
prefsForUser1.setValue(1, 4.5f);
... (8 more)
preferences.put(1L, prefsForUser1);   <-- 在输入中附上用户1的偏好

DataModel model = new GenericDataModel(preferences);
```


`GenericDataModel`使用了多少内存呢？被存储的偏好的个数决定了对内存的需求。根据一些经验性的测试得知，每个偏好大约消耗28字节的Java堆空间。这包括所有数据以及其他支持性的数据结构，如索引。如果你乐意，可以尝试：加载`GenericDataModel`，调用几次`System.gc()`，再比较`Runtime.totalMemory()`和`Runtime.freeMemory()`的结果。这种比较非常粗略，但会对数据所消耗的内存有一个合理的估计。

3.2.2 基于文件的数据

你通常不会直接使用`GenericDataModel`，而是借助于`FileDataModel`，后者从文件中读取数据，并将所得到的偏好数据存储到内存中，即存储到`GenericDataModel`中。

几乎任何正常的文件都可以用，比如第2章中采用的CSV（Comma-Separated Value，逗号分隔值）格式的那种文件。每行包含一个数据：用户ID、物品ID和偏好值。采用制表符分隔也是可以的。用zip或gzip压缩的文件同样可以，只要名字分别以.zip或.gz结尾。将数据以压缩格式存储是一个很好的主意，因为它会非常大且很容易压缩。

3.2.3 可刷新组件

虽然我们谈论的是加载数据，但仍有必要讲一讲*重加载数据*（reloading data），以及`Refreshable`接口，即在Mahout推荐程序相关类中所实现的几个组件。它只公开了一个方法`refresh(Collection<Refreshable>)`。该方法简单地请求组件在最新的输入数据上进行：重加载（reload）、重算（recompute）并刷新（refresh）自身状态，并事先让它的依赖（dependency）也这样做。

例如，`Recommender`在重新计算其内部数据索引时，多会在它所依赖的`DataModel`上调用`refresh()`。循环依赖（cyclical dependency）和共享依赖（shared dependency）被管理得很好，如图3-3所示（基于图2-2）。

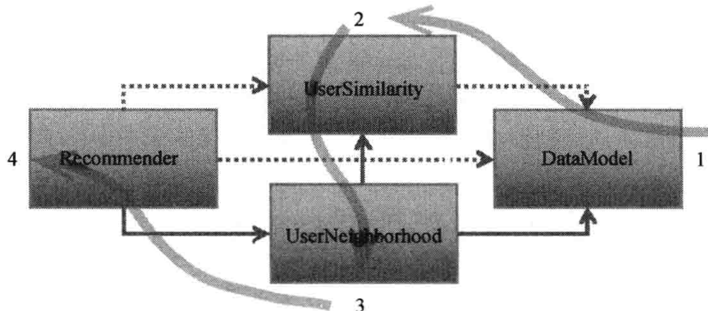


图3-3 一个简单的基于用户的推荐系统，箭头所示为组件之间刷新数据结构的顺序

注意，FileDataModel仅会在被请求的时候才从底层文件重新加载数据。出于性能考虑，它不会自动检测更新或定期重新加载文件的内容。这是refresh()方法该做的事。你大概不会只想刷新FileDataModel，而是希望所有依赖该数据的对象都被刷新。实际上，这就是为什么你总要在如下Recommender中明确调用refresh()的原因。

代码清单3-3　触发一个推荐系统的刷新

```
DataModel dataModel = new FileDataModel(new File("input.csv");
Recommender recommender = new SlopeOneRecommender(dataModel);
...
recommender.refresh(null);              <—— 刷新DataModel，然后刷新自己
```

因为规模是贯穿本书的主题，我们应该强调FileDataModel另一个有用的特性：更新文件。数据总在改变，通常改变的数据只是所有数据中很小的一部分，甚至可能仅仅是十亿个数据中的几个点。只为了几个数据的更新对一个包含十亿个偏好的数据做一个全新的复制，这是非常低效的。

3.2.4　更新文件

FileDataModel支持更新文件。它们就是在读取主数据文件之后额外生成的数据文件，并可以覆盖任何以前读取的数据。通过添加来形成新的偏好，还可以更新现有偏好。通过设一个偏好值为空的字符串来实现删除。

例如，考虑如下的更新文件：

```
1,108,3.0
1,103,
```

就是说，“更新（或生成）用户1对物品108的偏好，并将值设为3.0”，以及“删除用户1对物品103的偏好”。

这些更新文件必须和主数据文件在同一个目录下，且文件名的前缀（第一个域）相同。例如，如果主数据文件为foo.txt.gz，更新文件可为foo.1.txt.gz和foo.2.txt.gz。它们可以是压缩文件。

3.2.5　基于数据库的数据

有时数据就是太大了，无法放入内存。一旦数据集有几千万个偏好，内存需求会增长到几GB，在某些场景下可能无法支持这么大的内存容量。

偏好数据是有可能存储到一个关系数据库中并进行访问的，而Mahout支持这样做。在Mahout推荐程序中，一些类的实现出于性能考虑会把计算下放到数据库中。

要知道，当推荐引擎所用的数据来自数据库时，它的运行要比使用内存级的数据表示慢很多倍。这并不是数据库的错；通过合理地调优和配置，一个现代数据库可以用于极其高效地对信息进行索引和检索，但检索、整理（marshalling）、序列化（serializing）、传输和反序列化（deserializing）结果集的开销仍远大于从优化的内存级数据结构中读取数据的开销。由于推荐算法是数据密集型的，这种开销会快速积累。不过，当没有其他选择时，数据库仍是理想选择，或者虽然所用数据集不太大，但为了集成还需要重用一个现有的数据表，此时也应选择数据库。

3.2.6 JDBC和MySQL

偏好数据是通过JDBC访问的，使用了`JDBCDataModel`的实现。现在，`JDBCDataModel`的主要子类是为使用MySQL 5.*x*而写的：`MySQLJDBCDataModel`。它在MySQL的早期版本上也很好用，甚至可用于其他数据库，因为尽可能地使用了标准的ANSI SQL。变种的实现也不难，可以结合需求使用数据库所专有的语法和特性。

注意 在Mahout的开发版本中有一个专为PostgreSQL而做的`JDBCDataModel`的实现。还有一个`GenericJDBCDataModel`类，它允许你使用那些没有做专有实现的数据库中的数据。

默认情况下，这个实现假设所有的偏好数据位于一个名为`taste_preferences`的表中，其中用户ID的列为`user_id`，物品ID的列为`item_id`，偏好值的列为`preference`。其模式如表3-1[①]所示。该表还可以包含一个名为`timestamp`的字段，它的类型应该兼容于Java的`long`型。

表3-1 MySQL中`taste_preferences`表默认的模式

<table>
<tr><th>user_id</th><th>item_id</th><th>preference</th></tr>
<tr><td>BIGINT NOT NULL
INDEX</td><td>BIGINT NOT NULL
INDEX</td><td rowspan="2">FLOAT NOT NULL</td></tr>
<tr><td colspan="2">PRIMARY KEY</td></tr>
</table>

3.2.7 通过JNDI进行配置

`JDBCDataModel`实现还假设包含这个表的数据库可以通过一个`DataSource`对象来访问，这个对象已经注册到名为jdbc/taste的JNDI中。

你也许会问：什么是JNDI[②]？它的全称为Java Naming and Directory Interface，即Java命名与目录接口，它是J2EE（Java 2 Enterprise Edition）规范的核心。如果你正在一个Web应用中使用推荐引擎，并正在使用Tomcat或Resin这样的servlet容器，那么你很可能已经间接用到了JNDI。如果正通过容器（例如Tomcat的server.xml文件）配置数据库，你会发现这个配置通常会被JNDI中的`DataSource`所引用。

你可以将数据库配置为jdbc/taste，其中包含`JDBCDataModel`会使用的细节。这里有Tomcat可用配置的一个片段。

---

① 在MySQL中创建该表的命令可以写为：`CREATE TABLE taste_preferences (user_id BIGINT NOT NULL, item_id BIGINT NOT NULL, preference FLOAT NOT NULL, PRIMARY KEY (user_id, item_id), INDEX (user_id), INDEX (item_id))`。——译者注

② JNDI避免了数据库和程序之间的紧耦合。当数据库相关参数发生变更时，仅需在JNDI中修改相关配置，而无须修改程序。——译者注

代码清单3-4　在Tomcat中配置一个JNDI DataSource

```
<Resource
  name="jdbc/taste"
  auth="Container"
  type="javax.sql.DataSource"
  username="user"
  password="password"
  driverClassName="com.mysql.jdbc.Driver"
  url="jdbc:mysql://localhost:3306/mydatabase"/>
```

默认的名字（jdbc/taste）可以根据环境需要更换。你还可以像上面这样不去明确地为数据库和列命名。

3.2.8　利用程序进行配置

你也不必直接使用JNDI，而是将DataSource直接传递到MySQLJDBCDataModel的构造函数中。下面的代码清单显示了配置MySQLJDBCDataModel的一个完整示例，其中说明了如何使用MySQL Connector/J驱动（http://www.mysql.com/products/connector/），以及指定了表和列名的DataSource。

代码清单3-5　利用程序配置DataSource

```
MysqlDataSource dataSource = new MysqlDataSource ();
dataSource.setServerName("my_database_host");
dataSource.setUser("my_user");
dataSource.setPassword("my_password");
dataSource.setDatabaseName("my_database_name");
JDBCDataModel dataModel = new MySQLJDBCDataModel(
  dataSource, "my_prefs_table", "my_user_column",
  "my_item_column", "my_pref_value_column");
```

这就是将数据库中的数据用于推荐所需做的所有事情。

你现在已经得到了一个与所有推荐程序组件兼容的DataModel！但是正如MySQLJDBCDataModel的文档所说的，高效地推荐需要正确配置数据库与驱动。具体如下所述。

- 用户ID和物品ID列应为非空，而且必须被索引。
- 主键必须为用户ID和物品ID的组合。
- 列的数据类型根据Java中对应的long和float型来选择。在MySQL中，它们应为BIGINT和FLOAT。
- 注意调节缓冲区和查询高速缓存（query cache），见MySQLJDBCDataModel的Javadoc。
- 当使用MySQL的Connector/J驱动时，将驱动的参数（如cachePreparedStatements）设为true，细节同样见Javadoc。

上述讨论已经涵盖了使用Mahout推荐引擎框架中DataModel的基础。在这些实现中还有一个重要的变体需要讨论：如何表示偏好值缺失的数据。这听起来有些奇怪，因为偏好值似乎是推荐引擎所必须的输入数据。但有时，偏好值不存在或者忽略偏好值是有好处的。

3.3 无偏好值的处理

有时，输入推荐引擎的偏好没有值。也就是说，用户和物品是关联的，但是没有这种关联的强度描述。例如，一个新闻网站要根据用户以前浏览的新闻文章做推荐，但只知道一些用户和物品之间的关联，而没有更多的信息，因为用户通常不会去评价文章。用户在浏览文章之外甚至都很少会去做其他的事情。这时，我们仅能得知与用户关联的是哪篇文章，以及少量的其他信息。

在这里，我们别无选择；在输入中没有偏好值可以作为初始值。本章后续的技术和建议仍适用该场景。即便在输入中的确存在偏好值，有时忽略它们也会有好处。至少在有的时候，这样做没有坏处。

这并非要忘记用户和物品之间的关联，而是忽略其中的偏好强度。例如，当推荐一部新电影时，不是考虑你看过哪些电影以及你是如何评价它的，而只是简单地考虑你看过的是哪些电影。不是获取“用户1对电影103表达的偏好为4.5”，而是忘记4.5这个值，将“用户1和电影103有关联”这样的数据作为输入，这会很有用。图3-4说明了这种区别。

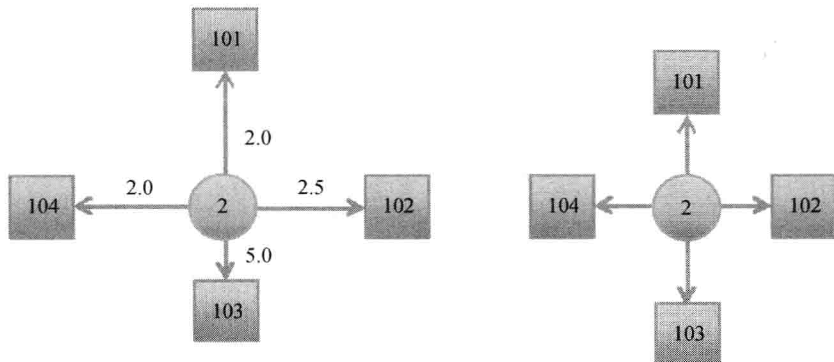


图3-4 用户和物品之间具有偏好值的关系（左图）和不具有偏好值的关系（右图）

由于缺少更好的术语来表达，在Mahout的语言里，这种没有偏好值的关联称为布尔型偏好（Boolean preference），因为一个关联只可能有两个值：存在或不存在。这并不意味着数据中的物品偏好是yes或no，而是会让用户–物品关联具有全部的三种可能状态：喜欢、不喜欢或无所谓。

3.3.1 何时忽略值


No. 3

为什么要忽略偏好值？因为这么做在一定场景下是有好处的，此时喜欢或不喜欢一个物品相对而言都差不多，至少和根本没有关联相比是这样的。

让我们举例说明。想象有这么一个人，他不喜欢古典作曲家Rachmaninoff的作品。事实上，他在自己的iTunes库中对Rachmaninoff的几个作品给出了1星或2星的评价。除了这些作品，世界上还有无数的音乐，其中一些是他从来没有听过的（就像挪威死亡金属乐，即Norwegian death

metal)。他甚至是因为足够了解Rachmaninoff而不喜欢他的作品，甚至在iTunes库的开头还留有几个Rachmaninoff的作品，这些都显示了他和这个作曲家的关联，甚至透漏出他对类似作品的一种偏好。与大千世界中他完全不知道的作品相比，这种关联是非常明显的。虽然他也许会给Rachmaninoff一个1星评价，而给Brahms一个5星，实际上这都传递了一些类似的信息：一种对古典音乐的兴趣。因此，忘记实际的评分反，认真思考这一事实，甚至可以给出更好的推荐。

你可能会反驳说这是用户的错误。难道他不会给Rachmaninoff一个4星？因为还有挪威死亡金属乐，而这可能才是他会给出1星评价的作品。也许如此，但这就是生活。输入常常是有问题的。你可能还会反驳说，虽然这对于从所有类型的音乐中做推荐是合理的，但忽略这些数据的话，在只推荐古典作曲家时可能使推荐效果变差。的确如此；但在一个领域中的好方案并不总是能移植到其他领域的。

3.3.2　无偏好值时的内存级表示

没有了偏好值会极大地简化偏好数据的表示，这会获得更优的性能并显著降低对内存的占用。如前所述，Mahout的`Preference`对象将偏好值存为4字节的Java `float`型。没有了偏好值，在内存中每个偏好能够节省4字节。实际上，重复前面的粗略测试可以看到，每个偏好的内存消耗平均减少了4字节，降为24字节。

这来自于对`GenericDataModel`孪生兄弟`GenericBooleanPrefDataModel`的测试。这是另一个内存级的`DataModel`实现，但其内部并不存储偏好值。它简单地将关联存为`FastIDSet`；例如，每个用户用1个，来代表与用户关联的所有物品ID。其中不包含偏好值。

因为`GenericBooleanPrefDataModel`也是一个`DataModel`，它有时可以代替`GenericDataModel`。`DataModel`的一些方法使用这个新的实现会更快，如`getItemIDsForUser()`，因为新的实现已经有现成的结果。有些则会变慢，如`getPreferencesFromUser()`，因为新的实现不使用`PreferenceArray`，必须实例化一个才能实现这个方法。

你也许想知道`getPreferenceValue()`会返回什么，因为这里并没有偏好值。它并不抛出`UnsupportedOperationException`，而会一概返回相同的假值：`1.0`。必须注意这一点，因为依赖于偏好值的组件仍会从该`DataModel`中获取一个值。这些偏好值是假值且不会改变，这会带来一些小问题。

让我们回到上一章的GroupLens示例。但代码改为使用`GenericBooleanPrefDataModel`，如代码清单3-6所示。

代码清单3-6　布尔型数据的生成与评估

```
DataModel model = new GenericBooleanPrefDataModel(
  GenericBooleanPrefDataModel.toDataMap(
    new FileDataModel(new File("ua.base"))));        <-- 使用GenericBooleanPrefDataModel

RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator();

RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
```

```
    public Recommender buildRecommender(DataModel model)
        throws TasteException {
      UserSimilarity similarity =
        new PearsonCorrelationSimilarity(model);
      UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(10, similarity, model);
      return
        new GenericUserBasedRecommender(model, neighborhood, similarity);
    }
  };

  DataModelBuilder modelBuilder = new DataModelBuilder() {
    public DataModel buildDataModel(
        FastByIDMap<PreferenceArray> trainingData) {
      return new GenericBooleanPrefDataModel(
        GenericBooleanPrefDataModel.toDataMap(
          trainingData));
    }                                         <- 构造一个
  };                                             GenericBooleanPrefDataModel

  double score = evaluator.evaluate(
    recommenderBuilder, modelBuilder, model, 0.9, 1.0);
  System.out.println(score);
```

该示例的关键在于`DataModelBuilder`。你可以用它控制评估过程构造用于训练数据的`DataModel`。`GenericBooleanPrefDataModel`获取输入的方式略为不同——通过一组`FastIDSet`而非`PreferenceArray`——有一个`toDataMap()`方法可以方便地转换它们。

阅读下一节之前，试着运行这段代码——它不会成功地结束。

3.3.3 选择兼容的实现

你会发现运行代码清单3-6中的代码会导致一个来自`PearsonCorrelationSimilarity`构造函数的异常`IllegalArgumentException`。初看这会让人感到奇怪：`GenericBooleanPrefDataModel`不也是一个`DataModel`吗？而且它除了不存储明确的偏好值之外，几乎与`GenericDataModel`相同。

如果缺少偏好值，像`EuclideanDistanceSimilarity`这样的相似性度量会拒绝工作，因为其结果会是未定义的（undefined）或无意义的，从而导致无用的结果。如果两个数据集是相同数值的简单重复，它们之间的皮尔逊相关系数是未定义的。这里，`DataModel`假设所有偏好值均为`1.0`。类似地，计算对应于空间上同一个点的所有用户之间的欧氏距离（Euclidean distance，又称欧几里得距离），即这里的(1.0, 1.0, …, 1.0)是无意义的，因为所有的相似性均为1.0。

注意 皮尔逊相关系数是两个数据集的协方差与其标准差之间的比值。当所有数据为1时，两个值均为0，而目前Java在计算0/0的相关结果时一定会返回“not a number”。[1]

---

① 在`PearsonCorrelationSimilarity`中通过`return Double.NaN;`来实现。——译者注

这个例子具有普遍意义，它说明了即使组件会采取一系列的标准接口来获得交互性，也无法保证每个实现都彼此相容。为了解决这个现实问题，需要一个合适的相似性度量。LogLikelihoodSimilarity就是这样的一个实现，因为它并非基于实际的偏好值。（我们稍后会讨论相似性度量。）用它来替代PearsonCorrelationSimilarity，结果为0.0。这很棒，因为这意味着完美的预测结果。是不是好得过火了呢？

很遗憾，的确如此。这个结果是当每个偏好值为1时，估计偏好和实际偏好之间的平均差值。结果自然会等于0；这个测试是无效的，因为它只能输出0。

但是查准率和查全率的评估仍是有效的。让我们尝试在下面的代码清单中来实现它。

代码清单3-7　利用布尔型数据评估查准率和查全率

```
DataModel model = new GenericBooleanPrefDataModel(
    new FileDataModel(new File("ua.base")));

RecommenderIRStatsEvaluator evaluator =
  new GenericRecommenderIRStatsEvaluator();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model) {
    UserSimilarity similarity = new LogLikelihoodSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(10, similarity, model);
    return new GenericUserBasedRecommender(
        model, neighborhood, similarity);
  }
};
DataModelBuilder modelBuilder = new DataModelBuilder() {
  @Override
  public DataModel buildDataModel(
      FastByIDMap<PreferenceArray> trainingData) {
    return new GenericBooleanPrefDataModel(
      GenericBooleanPrefDataModel.toDataMap(trainingData));
  }
};
IRStatistics stats = evaluator.evaluate(
    recommenderBuilder, modelBuilder, model, null, 10,
    GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD,
    1.0);
System.out.println(stats.getPrecision());
System.out.println(stats.getRecall());
```

所得查准率和查全率都是大约24.7%。这不算太好；回顾一下，这意味着返回的推荐中只有1/4是好的，而好的推荐中只有1/4在返回结果中。

这可追查出另一个问题；仍有一个地方隐藏着偏好值：GenericUserBasedRecommender。这个推荐程序仍基于其估计的偏好对推荐进行排序，但这些值均为1.0。因此顺序基本上是随机的。相反，你可以引入GenericBooleanPrefUserBasedRecommender（顾名思义）。这个变体可以让推荐形成更有意义的顺序。它为与其他类似用户相关的物品计算权重，用户相似度越高，这个权重越大。它并不生成加权平均。

尝试替代这个实现并重新运行代码。结果大约为22.9%——大致相同。这显然说明在这个数据上我们并未使用一个超高效的推荐系统。这里的目的并不是修复它，而仅仅为了审视如何在Mahout推荐程序中高效地部署布尔型数据。

还有其他DataModel的布尔型变种。FileDataModel会在输入数据不包含偏好值时（行只采用userID, itemID的形式），在内部自动使用GenericBooleanPrefDataModel。类似地，MySQLBooleanPrefDataModel适合在数据库表中无偏好值列时使用。否则它完全类似于MySQLJDBCDataModel。特别地，这种实现可以充分利用数据库中更多的快捷方式来提高性能。

最后，如果你想知道是否可以将布尔型和非布尔型数据在一个DataModel中混合使用，那么答案是：不行。一个解决办法是忽略偏好值，而将之视为布尔型数据。或者，如果你出于某种原因不希望抛弃它们，那些缺失的偏好值可以通过一些办法推测出来，即便只是简单地填充一个现有偏好值的平均数。

3.4 小结

在本章，我们探讨了如何在Mahout的推荐程序中表示偏好数据。其中涉及Preference对象，还有特定的数组和类似聚合的实现，如PreferenceArray和FastByIDMap。它们主要用来降低内存占用。

我们研究了DataModel，它是推荐程序输入的一个整体抽象。GenericDataModel把数据放在内存中，就像FileDataModel从文件中读取输入数据之后所做的一样。JDBCDataModel和其他实现支持基于关系库表的数据；我们特别观察了与MySQL的集成。

最后，我们查看了当输入数据仅包含用户–物品关联，而不包含偏好值时所带来的变化。有时，这就是可用的所有数据，而这必然减少对存储的需求。我们看到这类数据不兼容于PearsonCorrelationSimilarity等标准组件。我们还考虑如何解决这类问题并得到一个基于布尔型输入数据的函数级推荐程序。

下一章，我们会继续检视数据表示的各种可能并窥探其可用的推荐程序实现。

第4章

进行推荐

本章内容

- ❑ 进一步了解基于用户的推荐程序
- ❑ 相似性度量
- ❑ 基于物品的推荐程序及其他

我们用了一章的篇幅来讨论如何评价推荐程序，以及推荐程序的输入数据形式，是时候来深入探究推荐程序本身了。我们就此开始切入正题。

前面的章节提到两类典型的推荐算法，它们均在Mahout中得到实现：基于用户的推荐程序和基于物品的推荐程序。实际上，在第2章我们已经遇到过一个基于用户的推荐程序。本章将仔细探究和讨论这些算法背后的理论及其在Mahout中的实现。

两种算法均依赖于两个事物（用户或物品）之间的相似性度量，或者说等同性定义。相似性的定义有多种，本章将详细介绍Mahout中可供选择的方法。它们包括基于皮尔逊相关系数（Pearson correlation）、对数似然值（log likelihood）、斯皮尔曼相关系数（Spearman correlation）、谷本系数（Tanimoto coefficient）等的实现。

最终，本章还会介绍Mahout中实现的其他几种推荐算法，包括slope-one、基于SVD（SVD-based）和基于聚类（clustering-based）的推荐算法。

4.1 理解基于用户的推荐

如果你看过前面所讲的推荐算法，就会知道它是一种基于用户的推荐算法。它是在这个领域早期研究中阐述的方法，Mahout自然会有它的实现。“基于用户”这个说法有些不准确，因为所有推荐算法都建立在与用户和物品相关的数据上。基于用户的推荐算法的典型特征是，它建立在用户间有某种相似性的基础之上。事实上，这种算法在日常生活中很常见。

4.1.1 推荐何时会出错

你是否曾收到CD这样的礼物？我（Sean）在小时候从好心的成年人那里收到过。其中一个成年人走进当地的音乐商店并询问店员，于是有了下面的场景：

成年人：我要为一个男孩儿买张CD。
店　员：好的，他喜欢什么？
成年人：呃，现在的孩子都喜欢些什么？
店　员：他喜欢什么音乐或乐队呢？
成年人：对我而言，那些都太吵了，呃，我不知道。
店　员：嗯，好吧……我猜大多数年轻人都会购买New 2 Town这个男生组合的专辑。
成年人：就它了！

结果可想而知。不用说，他们送的礼物并非是我想要的。遗憾的是，这种基于用户进行推荐导致出错的事情比比皆是。但这种直觉还是对的：因为年轻人在音乐上的品味通常比较接近，一个年轻人很可能会喜欢其他年轻人追捧的专辑。根据人群之间的相似度进行推荐是非常合理的。

当然，推荐一个女孩儿们追捧的乐队专辑给男孩儿们可能并不合适。这里的问题在于相似性度量不再有效。是的，一群年轻人会有相对一致的品味：相对于柴迪科舞（zydeco）和古典音乐，流行音乐可能更受欢迎。但是，这种相似性太脆弱而难以为用：当把音乐作为推荐对象时，女孩儿们与男孩儿们并没有足够多的共性。

4.1.2 推荐何时是正确的

让我们回到前面的场景，来想象一个更好的情景：

成年人：我要为一个男孩儿买张CD。
店　员：他喜欢哪种音乐或者乐队？
成年人：我不知道，但他最好的朋友经常穿一件Bowling In Hades的T恤。
店　员：我知道，一个来自克利夫兰的非常流行的新金属乐队。我们正好有Bowling In Hades的最新专辑*Impossible Split: The Singles 1997–2000*。

这次好多了。这个推荐基于这样的假设，即两个好朋友在音乐上的品味会有些类似。相似性度量比较可靠时，结果就可能会更好。两个好朋友都喜欢Bowling In Hades的可能性比任意两个年轻人大得多。还有一些其他的途径能让结果更好：

成年人：我要为一个男孩儿买张CD。
店　员：他喜欢哪种音乐或者乐队？
成年人："音乐？"，哈，很好，我从他卧室墙上的海报里抄下了乐队名。The Skulks、Rock Mobster、Wild Scallions……你这里有吗？
店　员：我看看。这些专辑我的孩子也有一些。他总是在不停地谈论一些Diabolical Florist的新专辑，那么也许……

现在相似度的推断直接来自于对音乐的品味。因为其中所提及的两个孩子都喜欢一些相同的乐队，有理由相信他们都会喜欢对方的其他收藏。这比基于他们是好朋友来猜测他们的品味更为可靠。这种思路通过观察年轻人对音乐的品味来推断他们之间的相似度。这是基于用户的推荐系统最基本的逻辑。

4.2 探索基于用户的推荐程序

如果那两个人继续谈下去，可能还会得到更好的推测。为什么只根据一个孩子的音乐收藏来挑选礼物呢？何不多考虑几个类似的孩子？他们会留意哪些孩子更为相似（那些海报、T恤和散放在唱片机上的CD大多相同的），还会观察那些最相似的孩子都关注什么乐队，并据此来选择最合适的礼物。（然后，他们也许会成为Mahout的用户！）

4.2.1 算法

基于用户的推荐算法就来自这种直觉。下面是一个为用户（记为u）进行推荐的过程：

```
for（用户u尚未表达偏好的）每个物品i
  for（对i有偏好的）每个其他用户v
    计算u和v之间的相似度s
    按权重为s将v对i的偏好并入平均值
return值最高的物品（按加权平均排序）
```

外层循环简单地把每个已知物品（用户未对其表达过偏好的）作为候选的推荐项。内层循环逐个查看对候选物品做过评价的其他用户，并记下他们对该物品的偏好值。最终，将这些值的加权平均作为目标用户对该物品偏好值的预测。每个偏好值的权重取决于该用户与目标用户之间的相似度。与目标用户越相似，他的偏好值所占权重越大。

但是，每个物品都检查实在是太慢了。实际应用中，通常会先计算出一个最相似用户的邻域，然后仅考虑这些用户评价过的物品：

```
for每个其他用户w
  计算用户u和用户w的相似度s
  按相似度排序后，将位置靠前的用户作为邻域n
for（n中用户有偏好，而u中用户无偏好的）每个物品i
      for（n中用户对i有偏好的）每个其他用户v
  计算用户u和用户v的相似度s
    按权重s将v对i的偏好并入平均值
```

这一过程与前面的主要区别在于首先确定相似的用户，再考虑这些最相似用户对什么物品感兴趣。这些物品就成为推荐的候选项。后续的过程是一样的。这就是标准的基于用户的推荐算法，也是它在Mahout中的实现方式。

4.2.2 基于GenericUserBasedRecommender实现算法

本书最早的推荐程序示例展示了Mahout中一个实际的基于用户的推荐程序（代码清单2-2）。现在我们回顾一下它的构成，并对它的性能进行评估。

代码清单4-1　回顾一个简单的基于用户的推荐系统

```
DataModel model = new FileDataModel(new File("intro.csv"));
UserSimilarity similarity = new PearsonCorrelationSimilarity (model);
```

```
UserNeighborhood neighborhood =
  new NearestNUserNeighborhood (2, similarity, model);
Recommender recommender =
  new GenericUserBasedRecommender(model, neighborhood, similarity);
```

UserSimilarity封装了用户间相似性的概念，而UserNeighborhood封装了最相似用户组的概念。它们是标准的基于用户推荐算法的必要组件。

相似性的定义不是唯一的——前面选择CD的对话反映了在现实生活中人们关于相似性的几种看法。同样，最近邻用户也有多种不同的定义：最相似的5个，还是20个，还是所有相似度大于某一阈值的用户？为帮助理解这些选项，想象一下，假设你正在列一个婚礼客人的清单。你想邀请最亲密的朋友和家庭成员出席，但是你的朋友和家人远远超过预算允许的人数。为了决定邀请谁以及不邀请谁，你是不是会先定一个人数（比如50人），然后来选择最亲近的50个朋友或家人？50是一个合适的数字吗？40或100怎么样？或者，你会邀请每一个你认为很亲近的人吗？你会仅仅邀请真正的好朋友吗？谁会让你的婚礼派对非常成功？选择用户邻域与此类似。

引入新的相似性度量，结果就会发生显著变化。由此可知，提供推荐的方式是多种多样的——而这还只是调整了方法的一个侧面。Mahout是由多个组件混搭而成的，而非单一的推荐引擎，其各个组件的组合可以定制，从而针对特定应用提供理想的推荐。通常包括如下组件：

- 数据模型，由DataModel实现；
- 用户间的相似性度量，由UserSimilarity实现；
- 用户邻域的定义，由UserNeighborhood实现；
- 推荐引擎，由一个Recommender实现（此处为GenericUserBasedRecommender）。

要想推荐得更好更快，就必然需要经历一个漫长的试验和调优过程。

4.2.3 尝试GroupLens数据集

让我们回到GroupLens数据集，并将所用数据增加100倍。到http://grouplens.org下载包含1000万个评分的MovieLens数据集，目前它可以从地址http://www.grouplens.org/node/73获得。在本地解压后找到其中的ratings.dat文件。

出于某种原因，该数据的格式有别于之前的100 000评分数据集。其中ua.base文件可直接用于FileDataModel，但该数据集的ratings.dat文件则不能。简单的做法是使用标准的命令行文本处理工具将其转换为逗号分隔的形式，通常这也是最好的办法。专门编写代码来转换文件格式，或使用定制的DataModel，这不仅烦琐而且容易出错。

幸运的是，针对这个特例[①]还有一个更简单的办法。Mahout的示例模块（examples）包含了一个定制的GroupLensDataModel实现，它扩展了FileDataModel以读取这个文件。你需要确保这个代码在IDE项目的examples/src/java/main目录下。然后，如代码清单4-2所示的内容替换FileDataModel。

① 仅针对Gouplens数据集。——译者注

代码清单4-2　更新代码清单4-1来使用为GroupLens定制的DataModel

```
DataModel model = new GroupLensDataModel(new File("ratings.dat"));
UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
UserNeighborhood neighborhood =
  new NearestNUserNeighborhood(100, similarity, model);
Recommender recommender =
  new GenericUserBasedRecommender(model, neighborhood, similarity);
LoadEvaluator.runLoad(recommender);
```

运行这段代码，首先遇到的问题可能是OutOfMemoryError。在这里我们第一次碰到了规模问题。默认情况下，Java不会把堆（heap）大小设得过大。而这里，必须增加Java可用的堆空间。

这是一个探讨如何调节JVM来改善性能的好机会。可参考附录A更深入地了解JVM调优。

4.2.4　探究用户邻域

下面我们来评估推荐程序的精度。代码清单4-3再次给出了评估代码示例；在此之后，我们会认为你已经对它有了充分的了解，并可以独立构造和进行评估。

现在，我们尝试配置并调整该邻域的实现，如下面的代码清单所示。记住，这里使用的数据同样多出了100倍。

代码清单4-3　对这个简单的推荐引擎进行评估

```
DataModel model = new GroupLensDataModel (new File("ratings.dat"));
RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator ();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(
        DataModel model) throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(100, similarity, model);
    return new GenericUserBasedRecommender(
        model, neighborhood, similarity);
  }
};
double score = evaluator.evaluate(
    recommenderBuilder, null, model, 0.95, 0.05);
System.out.println(score);
```

注意，evaluate()的最后一个参数是0.05。这意味着仅有5%的数据用于评估。这纯粹是为了方便；评估是一个耗时的过程，使用全部数据会花上几个小时。为了快速评估变化，比较简便的做法是减小这个值。但是使用的数据太少可能会影响到评估结果的精度。参数0.95就是说使用95%的数据来构建要评估的模型，然后使用余下的5%来做测试。

代码运行所得到的结果可能会有出入，但约为0.89。

4.2.5 固定大小的邻域

此时，代码清单4-3中代码所给出的推荐来自于100个最相似用户构成的邻域（`NearestNUserNeighborhood`被设为邻域大小100）。推荐所依赖的最相似用户为100个，这个选择是随意的。如果选择10个会怎么样？推荐所依赖的相似用户虽然少了，但也会排除一些相似度较低的用户。包含3个最相似用户的邻域如图4-1所示。

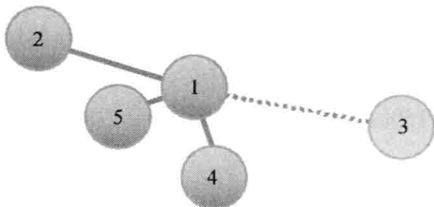


图4-1 通过确定最相似用户的数量来定义用户的邻域。这里，距离表示相似度：越远则越不相似。用户1的邻域由3个最相似的用户组成：5、4和2

尝试用10来替代100。推荐的评估结果，即估计值与实际偏好值的平均差异，为0.98左右。考虑到这一评估值越大越不好，这意味着选错方向了。最可能的解释是10个用户太少了。很可能后面的用户会有价值，如最相似的第11个、第12个用户等。他们不仅仍有很大的相似度，而且可能会关联到前10个用户没有涉及的一些物品。

尝试500个用户的邻域；结果降为0.75，这个结果自然较优。你可以多试一些值来为这个数据集找到最佳选项，但事实上并不存在一个万能的值；在真实数据上做一些试验对推荐程序的调优来说是很必要的。

4.2.6 基于阈值的邻域

假如不想用n个最相似用户构建邻域，那么如何直接选择那些很类似的用户并忽略其他人呢？你可以确定一个相似度阈值，并选择所有相似度超过这个阈值的用户。图4-2展示了一个基于阈值的用户邻域定义，你可将其与图4-1中的固定大小的邻域相对照。

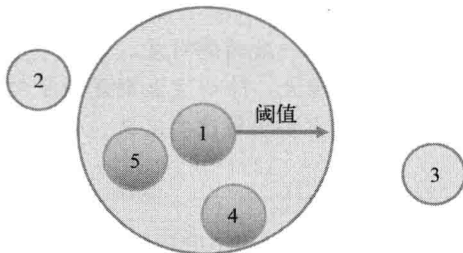


图4-2 通过一个相似度阈值来定义最相似用户的邻域

阈值应该设在-1和1之间，因为所有的相似性度量返回的相似度值都在该区间内。目前，我们的示例使用标准的皮尔逊相关系数作为相似性的度量标准。熟悉这种相关方法的读者应该知道0.7及以上的值意味着高度相关，可作为非常相似的一个合理定义。让我们改用ThresholdUser-Neighborhood。简单地修改一行来实例化ThresholdUserNeighborhood：

```
new ThresholdUserNeighborhood(0.7, similarity, model)
```

现在评估程序给推荐程序的评分为0.84。如果更严格的限定阈值，使用0.9会如何？评分为0.92，即性能更差了；前面的解释同样适用于此处——此阈值限定的邻域包含的用户数太少。如果设为0.5呢？评分变好了，可降至0.78。后面的示例会将这种邻域的阈值设为0.5。

现在，你可能想在真实数据上尝试更多的阈值以得到一个最优结果，不过我们通过简单的试验已经将估计精度提高了大约15%。

4.3　探索相似性度量

基于用户的推荐程序的另一个重要部分是UserSimilarity实现。基于用户的推荐程序非常依赖这个组件。如果对用户之间的相似性缺乏可靠并有效的定义，这类推荐方法是没有意义的。这也适用于基于用户的推荐程序的“近亲”——基于物品的推荐程序，它同样依赖于相似性。这一组件十分重要，我们将用接近本章1/3的篇幅来讨论标准的相似性度量及其在Mahout中的实现。

4.3.1　基于皮尔逊相关系数的相似度

到目前为止，示例都使用了PearsonCorrelationSimilarity这一实现，它是一个基于皮尔逊相关系数的相似性度量标准。

皮尔逊相关系数是一个介于-1和1之间的数，它度量两个一一对应的数列之间的线性相关程度。也就是说，它表示两个数列中对应数字一起增大或一起减小的可能性。它度量数字一起按比例改变的倾向性，也就是说两个数列中的数字存在一个大致的线性关系。当该倾向性强时，相关值趋于1。当相关性很弱时，相关值趋于0。在负相关的情况下——一个序列的值高而另一个序列的值低——相关值趋于-1。

注意　对于熟悉统计学的读者而言，皮尔逊相关系数是两个序列协方差与二者方差乘积的比值。协方差计算的是两个序列变化趋势一致的绝对量。当两个序列相对于各自的均值点向同一方向移动得越远，协方差值就越大。除以方差则是为了对这一变化进行归一化。使用Mahout中的皮尔逊相关系数并不需要理解这些定义，但如果你有兴趣，可以从网络上找到大量相关信息。

这一统计学中广泛使用的概念，同样可以用于度量用户之间的相似性。它度量两个用户针对同一物品的偏好值变化趋势的一致性——都偏高或都偏低。举个例子，再看看我们用过的第一个

样本数据文件intro.csv，如下所示。

代码清单4-4 一个简单的推荐系统输入文件

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

我们注意到用户1和5看起来相似，因为他们的偏好值好像在一同改变。对于物品101、102和103，他们大体达成一致：101最好，102略差，而103不理想。类似地，用户1和用户2则不那么相似。注意我们关于用户1的分析不包括物品104 ~ 106，因为用户1对它们的偏好是未知的。相似度的计算仅能在用户都表达了偏好的物品上进行。（在4.3.9节，我们将看到缺失偏好值的时侯该怎么进行推测。）

皮尔逊相关系数可表达这些相似性，如表4-1所示。这里不再重复计算的细节；可参考在线资源，如http://www.socialresearchmethods.net/kb/statcorr.php，了解相关系数计算的说明。

表4-1 在用户1和其他用户之间基于3个共有物品的皮尔逊相关系数

	物品101	物品102	物品103	与用户1的相关性
用户1	5.0	3.0	2.5	1.000
用户2	2.0	2.5	5.0	–0.764
用户3	2.5	—	—	—
用户4	5.0	—	3.0	1.000
用户5	4.0	3.0	2.0	0.945

* 用户与其自身的皮尔逊相关系数总是1.0。

4.3.2 皮尔逊相关系数存在的问题

尽管结果很直观，但某些情况下皮尔逊相关系数在推荐引擎中的表现会有点奇怪。

首先，它没有考虑两个用户同时给出偏好值的物品数目，在推荐引擎中这可能不太可靠。例如，两个看过200部相同电影的用户，即便他们给出的评分偶尔不一致，但可能要比两个仅看过两部相同电影的用户更相似。这在之前的数据中有所体现；注意，用户1和5对三个共同物品表达了偏好，他们的品位看似比较相近。但是，用户1和4的交集仅包含两个物品，却得到了1.0这个更高的相关值。这有点不符合常规。

其次，基于该计算的定义，如果两个用户的交集仅包含一个物品，则无法计算相关性。这也是没有计算用户1和3之间相关性的原因。在小的或稀疏的数据集上，这个问题就会凸现出来，因为其中用户的物品集很少重叠。当然，这可能也是一种优点：直观上讲，如果两个用户的交集仅有一个物品的话，他们可能并不太相似。

最后，只要任何一个序列中出现偏好值相同的情况[①]，相关系数都是未定义的（undefined）。这种情况并不需要两个序列中的偏好值都完全一样。例如，若用户5对所有三个物品的偏好值都是3.0，即使用户1有3.0以外的偏好值，也无法计算用户1与用户5之间的相似度（因为皮尔逊相关系数将是未定义的）。这一问题同样很可能出现在两个用户的偏好交集很小的情形。


No. 4

尽管皮尔逊相关系数在早期关于推荐系统的论文中很常见[②]，并且在很多介绍推荐系统的书中被提及，但它未必是最优的。当然，它也并不差；你只需要理解它是如何工作的。

4.3.3 引入权重

为了解决上述问题，`PearsonCorrelationSimilarity`在标准计算公式的基础上提供了一个扩展，即*加权*（weighting）。

皮尔逊相关系数并不直接反映其用到的物品数目，而我们是需要这个数字的。考虑的信息越多，所得的相关结果就越可靠。为了体现这一观点，最好在基于较多物品计算相关系数时，使正相关值向1.0偏移，而使负相关值向–1.0偏移。或者，当基于较少的物品计算相关系数时，可以让相关值向偏好值的均值偏移；这与前面的效果类似，但实现会较为复杂，因为它需要记录用户对的平均偏好值。

在代码清单4-3中，将值`Weighting.WEIGHTED`作为第二个参数传递给`PearsonCorrelationSimilarity`的构造函数即可实现上述方法。它会根据计算相关系数所用的数据点数，使偏好值向1.0或–1.0偏移。在这种情况下，重新运行前面的评估程序，可以看到分值有所改善，变为0.77。

---

① 此时该序列方差为0，导致皮尔逊相关系数计算公式中的分母为0。——译者注

② 附录C列出了一些相关文献。可重点参考Breese、Heckerman、Kadie的“Empirical Analysis of Predictive Algorithms for Collaborative Filtering”以及Herlocker、Konstan、Borchers和Riedl的“An Algorithmic Framework for Performing Collaborative Filtering”这两篇文章。

4.3.4 基于欧氏距离定义相似度

下面让我们尝试使用EuclideanDistanceSimilarity——将代码清单4-3中`UserSimilarity`的实现改为`new EuclideanDistanceSimilarity(model)`即可。

这一实现基于用户之间的*距离*。你可以将用户想象成多维空间中的点（维数等于总的物品数），偏好值是坐标。这种相似性度量计算两个用户点之间的欧氏距离d[①]。这个值本身并不代表相似度，因为该值越大表示距离越远，也就是说两个用户越不相似。用户越相似，这个值应该越小。因此，实际应用中取$1/(1+d)$为相似度。表4-2展示了一些示例。可以证明，距离为0（用户间的偏好完全相同）时，它的结果为1，而随着d的增加，会逐渐递减为0。这种相似性度量不会返回负数，而且值越大表示相似度越高。

4

表4-2 用户1与其他用户之间的欧氏距离及所得到的相似度评分

	物品101	物品102	物品103	距离	与用户1的相似度
用户1	5.0	3.0	2.5	0.000	1.000
用户2	2.0	2.5	5.0	3.937	0.203
用户3	2.5	—	—	2.500	0.286
用户4	5.0	—	3.0	0.500	0.667
用户5	4.0	3.0	2.0	1.118	0.472

在代码清单4-3改用`EuclideanDistanceSimilarity`后，得到结果0.75；比之前强一点，但差别不大。注意这里可以计算出任何用户之间的相似度，而皮尔逊相关系数就无法得出用户1与用户3之间的相似度。这算是欧氏距离的一个优点，但是根据一个共同物品得到的结果并不可靠。基于欧氏距离的实现同样可能得出一些与直觉不符的结果：用户1和用户4之间的相似度高于用户1和用户5。

4.3.5 采用余弦相似性度量

余弦相似性度量（cosine measure similarity）也将用户偏好值视为空间中的点，并基于此进行相似性度量。你需要将用户偏好值视为n维空间中的点。现在，假设有两条从原点——或者说$(0,0,\cdots,0)$——出发，分别到这两个点的射线。如果两个用户相似，则他们的打分也相似，也就是说他们的空间位置是很接近的，这样一来，至少这两条射线的方向也会差不多，两条射线之间的夹角会比较小。反之，如果两个用户不相似，则相应的两个点会相隔较远，从原点到这两点的射线很有可能指向不同的方向，形成的夹角会比较大。

与欧氏距离类似，这个夹角同样可以用来度量相似性。在这种情况下，夹角余弦代表相似度值。如果你对三角函数不熟悉，那么记住这点就行了：余弦取值范围在−1到1之间，小的夹角余弦接近1，大的夹角（接近180°）余弦接近−1。这个性质很好，因为小的夹角映射到了较高的相

① 回忆一下，欧氏距离就是各维坐标之差的平方和的平方根。

似度值，趋于1，而大的夹角则映射到–1附近。

你可能试图在Mahout中寻找类似CosineMeasureSimilarity的东西，但实际上它已经以一个意料之外的名字出现过了：PearsonCorrelationSimilarity。余弦相似性度量与皮尔逊相关系数并不是同一个东西，但如果有耐心做一些数学推理，你会发现当两个输入序列均值都为0（中心化）时，它们归结为同一个计算过程。因为在Mahout实现中会将输入中心化，因此这两个度量标准就变成一样的了。

余弦相似性度量在协同过滤中经常出现。你可以简单地通过PearsonCorrelationSimilarity来使用这一相似性度量。

4.3.6 采用斯皮尔曼相关系数基于相对排名定义相似度

对于我们来说，斯皮尔曼相关系数是皮尔逊相关系数的一个有趣的变体。该相关系数并非基于原始的偏好值，而是基于偏好值的*相对排名*来计算。想象一下，对于每个用户来说，他们偏好值最低的物品的偏好值被改为1。偏好值次低的物品偏好值被改为2，以此类推。类似于你对电影进行打分，给最不喜欢的电影一颗星，次不喜欢的电影两颗星，以此类推。然后，在变换后的偏好值上计算皮尔逊相关系数，这就是斯皮尔曼相关系数。

这个过程丢掉了一些信息。尽管它保留了偏好值最本质的东西——它们的顺序，但它最终也丢掉了用户对不同物品喜好程度的具体差异。很难说它是或不是一个好办法；它介于保留原始偏好值和将它们完全丢弃之间——这两种情况我们都已经讨论过了。

表4-3中是一些斯皮尔曼相关系数的计算结果。在这个已经很简单的数据集上，它本身的简单性导致了一些很极端的值：事实上，这里所有的相关系数都为1或–1，依赖于该用户与用户1偏好值的变化趋势是否一致。与皮尔逊相关系数一样，用户1和用户3之间没有相似度值。

表4-3 将偏好值变为排名，并得到用户1与其他用户之间的斯皮尔曼相关系数

	物品101	物品102	物品103	与用户1的相似度
用户1	3.0	2.0	1.0	1.0
用户2	1.0	2.0	3.0	–1.0
用户3	1.0	—	—	—
用户4	2.0	—	1.0	1.0
用户5	3.0	2.0	1.0	1.0

该方法由SpearmanCorrelationSimilarity实现。跟前面一样，你要将它放在评估代码中作为UserSimilarity使用。运行程序之后，就可以去喝杯咖啡休息一下。如果天黑了就上床睡觉。它不会很快运行结束的。这个实现非常慢，因为它需要做一些烦琐的工作来计算并存储排序结果。基于斯皮尔曼相关系数的相似性度量计算量很大，因此学术价值大于实用价值。当然，它对于一些小规模的数据集可能很有效。

借此机会正好介绍一下Mahout的缓存封装机制。CachingUserSimilarity是UserSimilarity的一种实现，它封装了另一个UserSimilarity的实现并缓存其结果。也就是

说它利用另一个实现进行计算，并将得到的结果进行内部缓存。然后，当需要提供一个已经计算过的用户间相似度时，它就可以直接返回，而不需要该实现重新进行计算。你可以用这个办法为任何相似性度量的实现添加缓存功能。当计算的代价很高时（例如在这里），引入这种机制是值得的。当然，缓存也有代价，它会消耗内存。

试试将SpearmanCorrelationSimilarity替换为下面的实现。

代码清单4-5 为UserSimilarity实现引入缓存机制

```
UserSimilarity similarity = new CachingUserSimilarity(
    new SpearmanCorrelationSimilarity(model), model);
```

建议将evaluate()函数的trainingPercentage从0.95升到0.99，从而让测试数据的规模从5%减为1%。将最后一个参数从0.05降为0.01，从而将评估比例从5%减为1%也不失为一个好办法[①]。这样一来，评估过程可以在大约几十分钟内结束。

结果可能在0.80左右。同样，这种相似性度量方法的优劣很难一概而论，但在这个特定的数据集上，它不如其他相似性度量方法有效。

4.3.7 忽略偏好值基于谷本系数计算相似度

有趣的是，还存在一些完全抛开偏好值的UserSimilarity实现。它们不管一个用户对一个物品的偏好值是高还是低，只关心用户是否表达过偏好。正如我们在第3章讨论过的，忽略偏好值并不会有太大影响。

TanimotoCoefficientSimilarity就是这样一个实现，它基于谷本系数。这个值也叫做Jaccard系数。它是由两个用户共同表达过偏好的物品数目除以至少一个用户表达过偏好的物品数目而得。如图4-3所示。

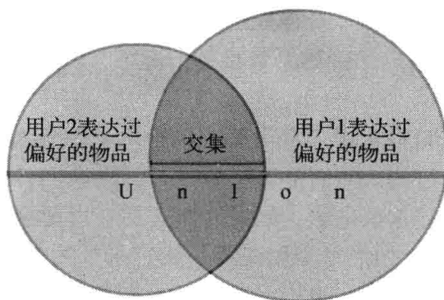


图4-3 谷本系数是两个用户各自表达过偏好的两个物品集合的交集，即重叠区域（深色区域）的大小，与并集（深色和浅色区域）大小的比值

换句话说，它是两个偏好物品集合的交集大小与并集大小的比值。它有如下性质：当两个用户的偏好集合完全重合时，结果为1.0。当他们没有任何共同点时，结果为0.0。结果永远不会为负，

① 前者调节各个用户偏好值中用来训练的比例，后者调节用于评估的用户数量。——译者注

但也没有关系。用一些简单的数学变换就可以把结果变换到−1到1之间：相似度 = 2 × 相似度 −1。对于整个框架来说，这不会有太大影响。

表4-4列出了用户1与其他用户之间的一些基于谷本系数的相似度值。

表4-4　基于谷本系数计算得到的用户1与其他用户之间的相似度值。注意偏好值本身被忽略了，因为计算过程并没有用到它们

	物品101	物品102	物品103	物品104	物品105	物品106	物品107	与用户1的相似度
用户1	×	×	×					1.0
用户2	×	×	×	×				0.75
用户3	×			×	×		×	0.17
用户4	×		×	×		×		0.4
用户5	×	×	×	×	×	×		0.5

注意这一相似性度量并不仅仅取决于用户共同表达过偏好的物品，它同时也需要考虑仅有一个用户表达过偏好的物品。因此，跟前面不同，所有的7个物品都出现在计算过程中。

当且仅当偏好值为布尔值或者根本没有偏好值可用时，你才需要用到这一度量方法。如果有偏好值，而且偏好值中除了噪声还有更多的信息，或许你也会使用这一方法。但是，此时使用那些基于具体偏好值的度量方法往往效果会更好。在GroupLens数据集上，使用谷本系数后分值变差了一些，上升至0.82。

4.3.8　基于对数似然比更好地计算相似度

尽管看起来不像，但基于对数似然比的相似度（Log-likelihood–based similarity）实际上类似于基于谷本系数的相似度。它是另一种不考虑具体偏好值的度量方法。计算这一相似度所需要的数学知识已经超出了本书的讲述范围。与谷本系数类似，它也基于两个用户共同评估过的物品数目，但在给定物品总数和每个用户评价物品数量的情况下，其最终结果衡量的是两个用户[①]有这么多共同物品的“不可能性”。

考虑两个电影爱好者，各自都看了一些电影，也给出了评分，但仅仅有《星球大战》和《卡萨布兰卡》这两部是其共同看过的。他们是否相似呢？如果他们各自都看了几百部电影，这一点就没有多大意义了。很多人看过这些电影，如果这两人都看过很多电影，但仅有这两部是都看过的，那他们可能并不相似。反之，如果每个用户都只看过很少的电影，这两部却都是他们看过的，则可能暗示他们在电影方面爱好相似；此时重叠占的比重很大。

谷本系数已经能够反映这一思想，因为它考虑了两者交集大小与并集大小的比值。对数似然比的计算则略有不同。它试图反映两个用户由于机缘巧合发生重叠的不可能性。也就是说，两个不相似的用户毫无疑问会共同评价一些电影，但是两个相似用户之间的重叠不太可能是出于巧合。通过一些统计检验，这一相似性度量试图判断两个用户口味不相似的不可能性有多大；不可

① 不仅仅是针对不相似用户的衡量。——译者注

能性越大，两个用户的相似度越高。最终的相似度值可以解释为发生重叠的非偶然概率。

在我们的小数据集上计算出来的一些相似度值如表4-5所示。正如你所见，在这种度量方式下，用户1与其自身或其他任何有共同偏好的用户之间的相似度都不等于1.0。同前面类似，要使用基于对数似然比的相似性度量，只需将`new LogLikelihoodSimilarity`插入到代码清单4-3中。

表4-5 用户1与其他用户之间的相似度，基于对数似然比相似性度量

	物品101	物品102	物品103	物品104	物品105	物品106	物品107	与用户1的相似度
用户1	×	×	×					0.90
用户2	×	×	×	×				0.84
用户3	×			×	×		×	0.55
用户4	×		×	×		×		0.16
用户5	×	×	×	×	×	×		0.55

虽然很难一概而论，基于对数似然比的相似度往往优于基于谷本系数的相似度。从某种意义上说，它是一个更智能的度量标准。运行评估程序就可以看到，至少对于这个数据集和推荐程序，相比于`TanimotoCoefficientSimilarity`，它会将性能改善为0.73。

4.3.9 推测偏好值

有时候数据过少会成为问题。例如，在某些情况下一些用户对只在一个物品上重叠，皮尔逊相关系数无法计算任何相似度。皮尔逊相关系数也不考虑只有一个用户表达过偏好的物品。

如果在所有缺失的数据点上填充一个默认值会怎样呢？例如，通过为用户没有评价过的物品推测一个默认偏好值，系统可以认为每个用户都评估过所有的物品。可以通过`PreferenceInferrer`接口引入这种机制，目前可用`AveragingPreferenceInferrer`这一实现，它为每个用户计算其已评估物品的平均值，并以此作为未评估物品的偏好值。在`UserSimilarity`实现中调用`setPreferenceInferrer()`方法可以开启这个选项。

尽管提供了这一机制，但实际应用中它并不是很有效。之所以提到它，是因为关于推荐系统的早期研究中提到了它。理论上讲，完全基于已有信息来填补缺失信息并不会增加任何信息量，相反，这显然会严重拖慢计算速度。它可以用于实验，但在真实数据集上恐怕不会起到什么作用。

你现在已经了解了Mahout中所有关于用户间相似性度量的实现。掌握这些知识将会使你事半功倍，因为Mahout中的物品间相似性度量的实现与此非常类似。也就是说，同样的计算可以用于定义物品之间的相似性，而不仅仅是用在用户上。现在及时补充这些知识是很有必要的，因为相似性的概念同样也是你将遇到的另一类Mahout推荐程序——基于物品的推荐程序——的基础。

4.4 基于物品的推荐

我们已经了解了Mahout中基于用户的推荐程序；不只是一个推荐程序，而是建立在基于用户的推荐方法上的许多工具，是通过搭配多种多样的组件来实现的。

顺其自然，下面我们将要讨论基于物品的推荐程序（item-based recommender）。由于前面讨论过的很多组件（数据模型、相似性度量的实现）也适用于基于物品的推荐，因此这一节会简短一些。

基于物品的推荐是以物品（而不是用户）之间的相似度为基础的。在Mahout中，这意味着基于`ItemSimilarity`实现相似性度量，而非基于`UserSimilarity`。为了说明这一点，我们回顾音乐商店里的场景，那两个人还在努力尝试着为那个男孩儿推荐他喜欢的专辑。想象一下，他们现在从另一个角度来推测那个男孩儿的喜好：

> 成年人：我要为一个男孩儿买张CD。
>
> 店　员：好的，他喜欢什么音乐或乐队呢？
>
> 成年人：他总是穿一件Bowling In Hades的T恤，好像还有这个乐队所有的专辑。你有什么要推荐的吗？
>
> 店　员：啊，喜欢Bowling In Hades的人大都喜欢*Rock Mobster*这个新专辑。

这个方法看起来不错。它跟前面的例子也是有区别的。唱片商店的店员根据男孩儿喜欢的东西，为他推荐了一个类似的专辑。这与以前是不同的，以前的问题是："谁与这个男孩儿相似？他们又喜欢什么呢？"现在的问题是："什么东西与男孩儿喜欢的东西类似？"

图4-4显示了基于用户和基于物品的推荐程序之间的本质区别。它们通过不同的途径选择要推荐的物品：分别是通过相似的用户和相似的物品。

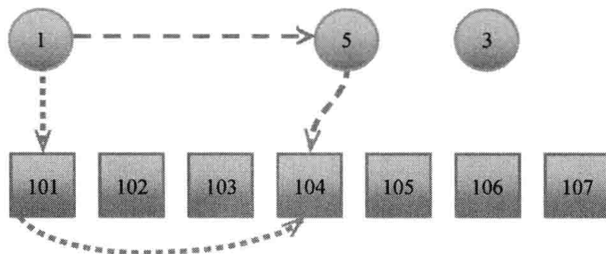


图4-4　基于用户和基于物品的推荐之间的区别：基于用户的推荐（长虚线）寻找相似的用户，并了解他们喜欢什么；基于物品的推荐（短虚线）了解用户的喜好，并寻找相似的物品

4.4.1　算法

了解基于用户的推荐程序之后，你会觉得这个算法很熟悉。这也是它在Mahout中的实现方式。

```
for (用户u尚未表达偏好的) 每个物品i
    for (用户u表达偏好的) 每个物品j
        计算i和j之间的相似度s
        按权重为s将u对j的偏好并入平均值
return  值最高的物品 (按加权平均排序)
```

第三行显示了它基于物品之间的相似度，而非像前面那样基于用户间的相似度。两种算法比

较相似，但也不完全是彼此相同的。它们有一些显著的区别。例如，基于物品的推荐程序运行时间随着物品的个数增长，而基于用户的推荐程序运行时间随着用户数增长。

这也成为使用基于物品的推荐程序的一个理由：如果物品数比用户数少很多的话，基于物品的推荐程序会带来显著的性能提升。

此外，物品要比用户稳定一些。像DVD这样的物品，我们可以合理的假设随着时间的不断推移，搜集到的数据越来越多，对物品之间相似度的估计值会趋于收敛。它们没有理由剧烈或频繁地发生变化。类似的现象也许对用户也是如此，但随着时间推移，用户会有一些新的认识，接触到新的信息，因此用户的喜好也会随之发生改变。结合前面的例子来看，*Bowling In Hades*和*Rock Mobster*这两个专辑一年后的相似度可能与今天差不多。但前面提到的同一群粉丝一年后的品味仍然相似的可能性就很小了，因而他们之间的相似度也会发生改变。

如果物品之间有更稳定的相似度，那么它们就更适合于预先计算。预先计算相似度有一定的工作量，但它大大提升了运行时的推荐效率。在运行时需要快速提供推荐结果的场合，这个特性是很有意义的，例如一个新闻网站，它必须及时地将每个新闻视图推荐出去。

在Mahout中，`GenericItemSimilarity`类可以用来预先计算并存储`ItemSimilarity`的结果。它可以用于你所见过的任何实现中，只要你愿意，就可以把它加到之后的代码片段中。

4.4.2 探究基于物品的推荐程序

现在我们将一个简单的基于物品的推荐程序嵌入到评估框架中，代码如下。此时的程序使用`GenericItemBasedRecommender`来取代`GenericUserBasedRecommender`，并且它的依赖项更为简洁。

代码清单4-6 一个基础的基于物品的推荐程序的核心部分

```
public Recommender buildRecommender(DataModel model)
    throws TasteException {
  ItemSimilarity similarity = new PearsonCorrelationSimilarity(model);
  return new GenericItemBasedRecommender(model, similarity);
}
```

`PearsonCorrelationSimilarity`在此处仍然是有效的，因为它也实现了一个与`UserSimilarity`接口完全类似的`ItemSimilarity`接口。这里相似性的定义与前面相同，都是基于皮尔逊相关系数的，只不过现在是在物品而不是用户之间度量相似性。也就是说，它比较的是由许多用户针对一个物品所给出的偏好值序列，而不是一个用户针对许多物品的偏好值序列。

`GenericItemBasedRecommender`比较简单，它仅仅需要一个`DataModel`和一个`ItemSimilarity`——没有`ItemNeighborhood`。你可能奇怪它为什么跟`GenericUserBasedRecommender`不对称。回想一下，基于物品的推荐过程并非从零开始：已经有一些用户表达过偏好的物品。这与基于用户的推荐程序在第一步确定的相似用户邻域是类似的。计算各个用户偏好物品的邻域对该算法后面的步骤没有任何意义。

建议你尝试不同的相似性度量，就像之前针对基于用户的推荐程序所做的尝试一样。并非所

有的UserSimilarity实现都有相应的ItemSimilarity实现。现在，你已经掌握了使用不同相似性度量标准时，在GroupLens数据集上评估基于物品的推荐程序准确性的方法。为方便起见，结果重新计算后列于表4-6中。

表4-6 使用不同ItemSimilarity度量标准的评估结果

实　现	相 似 度
PearsonCorrelationSimilarity	0.75
PearsonCorrelationSimilarity +权重	0.75
EuclideanDistanceSimilarity	0.76
EuclideanDistanceSimilarity +权重	0.78
TanimotoCoefficientSimilarity	0.77
LogLikelihoodSimilarity	0.77

你可能会注意到，这种推荐程序的运行速度要比以往快得多。这并不奇怪，假设数据集中有70 000个用户和10 000个物品。在物品数小于用户数的情况下，基于物品的推荐程序通常会运行得更快。你可能希望将用于评估的数据比例提升至20%左右（将0.2作为最后一个参数传给evaluate()）。这样会得到一个更可靠的评估结果。可以看到，对于这个数据集，这些实现所得到的结果之间并没有明显的区别。

4.5 Slope-one 推荐算法

你喜欢《情枭的黎明》（*Carlito's Way*）这部电影吗？很多喜欢这部电影的人也会喜欢Al Pacino的另一部电影《疤面煞星》（*Scarface*）。但人们往往更喜欢后者。我们可以想象如果一个人给《情枭的黎明》四颗星，那么他可能会给《疤面煞星》五颗星。所以如果你给《情枭的黎明》三颗星，我们猜你可能会给《疤面煞星》四颗星——比《情枭的黎明》多一颗星。

如果你同意上面的推理，你就会喜欢slope-one推荐算法（http://en.wikipedia.org/wiki/Slope_One）。它基于新物品与用户评估过的物品之间的平均偏好值差异来预测用户对新物品的偏好值。

举个例子，我们假设人们给《疤面煞星》的平均分要比《情枭的黎明》高出1.0分。再假设，平均看来，人们对《疤面煞星》的评价与《教父》（*The Godfather*）相同。现在，我们有一个用户给《情枭的黎明》打了2.0分，给《教父》打了4.0分。怎样合理的估计他对《疤面煞星》的评价呢？

根据《情枭的黎明》，较为理想的估计应该是2.0+1.0=3.0。根据《教父》，又应该是4.0+0.0=4.0。二者的均值可能是一个更好的估计：3.5。这就是slope-one推荐方法的基石。

4.5.1 算法

我们认为两个物品的偏好值之间存在着某种线性关系，所以可以通过某个线性函数，例如$Y=$

$mX+b$，由物品X的偏好值估计出物品Y的偏好值。Slope-one推荐算法正是基于这一假设来运作，并因此而得名。Slope-one推荐程序做了进一步的简化假设$m=1$，即斜率为1。现在我们只需要确定$b=Y-X$，即物品两两之间偏好值的（平均）差异。

这意味着算法需要有一个重要的预处理步骤，即完成所有物品对之间偏好值差异的计算：

```
for每个物品i
  for每个其他物品j
    for对i和j均有偏好的每个用户u
      将物品对（i与j）间的偏好值差异加入u的偏好
```

在此基础上，可得最终的推荐算法如下：

```
for用户u未表达过偏好的每个物品i
  for用户u表达过偏好的每个物品j
    找到j与i之间的平均偏好值差异
    添加该差异到u对j的偏好值
    添加其至平均值
Return值最高的物品（按平均差异排序）
```

我们在本书前面例子中用到的小数据集上求平均差异值，并将其列在表4-7中。

表4-7 所有物品对的平均偏好值差异。对角线上的单元格值为0.0。下半部分单元格的值为其关于对角线对称的单元格值的相反数，所以这一部分也没有在表中列出。某些差异值不存在，例如102~107，因为没有用户同时对物品102和物品107给出偏好值

	物品101	物品102	物品103	物品104	物品105	物品106	物品107
物品101		–0.833	0.875	0.25	0.75	–0.5	2.5
物品102			0.333	0.25	0.5	1.0	—
物品103				0.167	1.5	1.5	—
物品104					0.0	–0.25	1.0
物品105						0.5	0.5
物品106							—
物品107							

Slope-one的吸引力在于其算法的在线部分执行很快。与基于物品的推荐程序类似，它的性能不受数据模型中用户数目的影响。它仅仅依赖于物品之间偏好值的平均差异，而这些差异值可以预先计算好。另外，它的底层数据结构更新的效率很高：当一个偏好值发生了改变，只需要更新相关的差异值。在偏好值变化频繁的场合，这是一个优点。

注意，存储所有物品对之间的偏好值差异所需要的内存随物品数的平方增长。2倍的物品数意味着4倍的内存用量！

4.5.2 Slope-one实践

使用Slope-one推荐程序很容易。它不再必须使用相似性度量标准这个参数：

```
new SlopeOneRecommender(model)
```

再一次在GroupLens的一千万项评价的数据集上运行我们的标准评估程序，结果会在0.65左右。这已经是最好的了。实际上，简单的Slope-one方法在很多情况下都有着不错的表现。与我们目前为止接触过的其他推荐方法不同，它不再需要用到相似性度量标准。参数调试的工作量大大减少。

与皮尔逊相关系数类似，简单形式的slope-one算法也存在一个弱点：物品之间的差异被赋予了相同的权重，而没有考虑这些差异的可靠性——计算这些差异时使用了多少数据。我们假设仅有一个用户同时对《情枭的黎明》和《恋恋笔记本》（*The Notebook*）这两部电影给出了评价。这是可能的，因为这两部电影风格迥异。这两部电影之间的差异很容易计算，但这是不是跟基于数千名用户计算出来的《情枭的黎明》和《教父》之间的差异一样有用呢？看起来不是。后者可能更可靠，因为它是在更多用户基础上计算的平均值。

与前面类似，引入某种形式的权重有助于改善这些推荐算法的性能。`SlopeOneRecommender`提供了两种权重：基于数量的和基于标准差的权重。回忆一下Slope-one估计偏好值的过程，它把用户所有已评估物品的偏好值加上一个差异，然后将这些结果取平均作为最后的估计值。基于数量的加权会在那些基于更多数据算出的差异值上加上更大的权重。平均值变为加权平均值，其权重就是差异数量——计算某一差异时用到的用户数量。

类似地，基于标准差的加权通过偏好值差异的标准差来计算权重。较低的标准差有较高的权重。如果两部电影偏好值的差异对于很多用户都是一致的，那么它可能更可靠，因而会得到更高的权重。如果它在不同用户之间差异很大，那它就不那么重要了。

这些变种效果都很不错，所以它们默认是被开启的。运行前面的评估程序时，你已经用过这种方法了。这里，我们关掉它们再看看效果。

代码清单4-7　选用不加权的`SlopeOneRecommender`

```
DiffStorage diffStorage = new MemoryDiffStorage(
    model, Weighting.UNWEIGHTED, Long.MAX_VALUE));
return new SlopeOneRecommender(
 model,
 Weighting.UNWEIGHTED,
 Weighting.UNWEIGHTED,
 diffStorage);
```

结果是0.67——在这个数据集上只差了一点点。

4.5.3 `DiffStorage`和内存考虑

slope-one是有代价的：内存消耗。事实上，如果你用10%的数据（大概100 000个评价）来进行评估，1 GB的堆空间都不够用。差异值使用很频繁，而且它们的计算代价很高，因此需要预先计算并保存。但是将它们全都保存在内存中代价是很高的，有必要把这些差异值存到别的地方。

幸运的是，我们有一些像`MySQLJDBCDiffStorage`这样的实现可以满足这一需求，它们允许预先计算差异值并在数据库中更新它们。它们需要和JDBC支持的`DataModel`实现（比如`MySQLJDBCDataModel`）结合起来使用，示例如下。

代码清单4-8 创建一个JDBC支持的DiffStorage

```
AbstractJDBCDataModel model = new MySQLJDBCDataModel();
DiffStorage diffStorage = new MySQLJDBCDiffStorage(model);
Recommender recommender = new SlopeOneRecommender(
    model, Weighting.WEIGHTED, Weighting.WEIGHTED, diffStorage);
```

在MySQLJDBCDataModel中，MySQLJDBCDiffStorage使用的表名和列名可以通过构造器参数（constructor parameter）来定制。

4.5.4 离线计算量的分配

预先计算物品之间的差异值是一个很重要的工作。你首先遇到的问题可能是无法满足生成大量数据所带来的内存需求，随之而来的就会是计算这些差异值所耗费的时间，你可能会考虑是否有办法通过分布式计算提高速度。在运行时，如果有新的信息到来，差异值很容易更新，因此预先的离线计算相对不是很频繁，放在这种（分布式）模式下是可行的。

Mahout支持通过Hadoop分布式计算差异值。第6章会介绍Mahout支持的所有Hadoop相关的推荐算法，从而更深入地探索这一过程。

4.6 最新以及试验性质的推荐算法

Mahout也包含一些其他推荐方法的实现。本节简要介绍了三个比较新的实现，它们可能还在不断的改进之中，也有些是最近的一些试验性质的实现。这些思路都可能有实用价值或值得进一步改进。

4.6.1 基于奇异值分解的推荐算法

这里面最有意思的实现莫过于基于SVD（Singular Value Decomposition，奇异值分解）的SVDRecommender了。这是从线性代数引入到机器学习中的一项重要技术。完全理解它需要一些高阶的矩阵代数知识，并要求对矩阵分解有比较深入的理解，但对于SVD在推荐系统中的应用来说，这些不是必要的。

为了对SVD在推荐系统中的作用有一个直观的理解，我们假设你询问一个朋友她喜欢什么类型的音乐，然后她列出了如下的艺术家：

- Brahms
- Chopin
- Miles Davis
- Tchaikovsky
- Louis Armstrong
- Schumann
- John Coltrane
- Charlie Parker

她可能也总结了一下，表示自己喜欢古典音乐和爵士乐。这种表述传达的消息就不那么精确了，但也不是太不精确。不管基于哪种表述，你都可能（正确地）推断出，相对于古典摇滚乐团Deep Purple，她可能更喜欢贝多芬。

当然，推荐引擎处理的是具体的数据点，而不是笼统的概念。其输入是用户对很多特定物品的偏好值——可能是如前面所述的艺术家列表，而不是后来的概要描述。考虑到性能，处理更小的数据集可能是一个不错的选择。例如，如果iTunes的Genius基于数百万个流派，而不是基于数十亿首独立的歌曲评分来进行推荐，就能运行得更快；而且，就音乐推荐而言，效果并不会差太多。

这里，SVD就能起到上述的提炼作用。它从用户对各个物品的偏好值中提炼出数量较少但更具一般性的特征（例如流派）。这可能是一个小得多的数据集。

尽管这一过程丢掉了一些信息，某些时候却能改善推荐结果。这一过程有效地平滑了输入数据。例如，假设有两个汽车发烧友，一个喜欢克尔维特（Corvettes），另一个喜欢科迈罗（Camaros），他们都想得到对汽车的推荐。这两个发烧友品味比较接近：他们都喜欢雪佛兰跑车。但在针对此类问题的典型数据模型中，这两辆车是不同的物品。如果偏好值上没有任何重叠的话，这两个用户可能被认为是不相关的。然而，基于SVD的推荐方法却可能找到这种相似性。SVD的输出可能包含一些对应于雪佛兰或跑车这类概念的特征，通过这些特征就可以把这两个用户联系起来。根据特征的重叠就可以计算出某种相似度。

使用SVDRecommender很简单，代码如下：

```
new SVDRecommender(model, new ALSWRFactorizer(model, 10, 0.05, 10))
```

SVDRecommender使用了一个Factorizer来完成这项工作；首次使用可以试试上面的ALSWRFactorizer。这里我们不展开讨论Factorizer的选择。

第一个数值参数是SVD最终要生成的特征数目。这里没有标准答案；它可以是你从某人音乐品味中归结的流派个数，例如本节中的第一个例子。第二个参数是λ，它控制着一个叫正则化的分解器（factorizer）特征。最后一个参数是需要执行的训练步骤数。你可以认为它控制了花在归纳上的时间，而较大的值意味着更久的训练时间。

这个方法可以得到不错的结果（在GroupLens数据集上是0.69）。目前，这一实现的主要问题是需要在内存中完成计算。整个数据集都需要放在内存中，如果不满足这个需求，却恰恰使该技术有了用武之地，因为它可以在不显著降低输出质量的情况下缩减输入规模。以后，这个算法会基于Hadoop重新实现，使得SVD可以将庞大的计算分配到多台机器上，但这在当前的Mahout中尚不可用。

4.6.2　基于线性插值物品的推荐算法

线性插值不同于以往的基于物品的推荐方法，它在Mahout中的实现是KnnItemBasedRecommender。Knn是k nearest neighbors的简称，也用于NearestNUserNeighborhood中。正如你之前在图4-1中所见到的，UserNeighborhood的实现选择指定个数的最相似用户作为近似用户邻域。这里的线性插值算法也使用了用户邻域的概念，但采用了另一种方式。

KnnItemBasedRecommender仍然通过用户已评估过的物品的加权平均来估计偏好值，但权重不再是相似度，而是用一些线性代数技术计算出的所有物品对之间的最优权重集合——这就是

用到线性插值的地方。自然，此时就可以通过一些数学技巧对权重进行优化。

在现实中，在所有物品对上都做这个计算的代价是很高的，所以我们预先计算出与目标物品（需要估计偏好值的物品）最相似的物品的邻域。即选定*n*个最接近的邻点，就像NearestNUserNeighborhood所做的那样。你可以尝试如下代码清单中的KnnItemBasedRecommender。

代码清单4-9 部署KnnItemBasedRecommender

```
ItemSimilarity similarity = new LogLikelihoodSimilarity(model);
Optimizer optimizer = new NonNegativeQuadraticOptimizer();
return new KnnItemBasedRecommender(model, similarity, optimizer, 10);
```

这段代码使用对数似然相似性度量标准计算出10个最近邻物品。然后，它将会使用二次规划（quadratic programming）技术来求解——计算线性插值的基本方法。这一计算的细节不在本书的讨论范围之内。

这一实现是很有用的，但其当前版本即便在中等规模的数据集上运行也比较缓慢。可以认为它适用于小数据集，或者适用于学习和扩展。在GroupLens数据集上，其评估结果为0.76。

4.6.3 基于聚类的推荐算法

基于聚类的推荐被认为是基于用户推荐程序的最好变种。它将物品推荐给相似用户簇，而不是具体用户。它需要一个将所有用户划分到不同簇的预处理过程。然后，它为每个簇提供推荐，这样，推荐的物品就会被尽可能多的用户接受。

这种方法的好处在于运行时的推荐很快，因为几乎一切都预先计算好了。或许这种方式给出的推荐不够个性化，因为推荐是为一个群组而不是个人提供的。对于几乎没有历史偏好数据的新用户而言，用这种方法提供推荐可能会更有效。只要用户可以合理归入一个相关的簇，推荐结果就会随着对用户的不断了解而越来越好。

该算法得名于它循环地将最相似的簇拼接成更大的簇，而这潜在地将用户组织成某种层次结构，或者说树形结构，如图4-5所示。

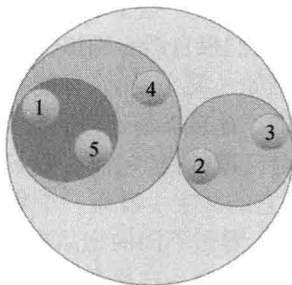


图4-5 聚类示意图。用户1和用户5首先被聚集到一起，还有2和3，因为他们是最相近的。然后，4被并入1和5的簇中从而得到一个更大的簇，朝着树形结构又推进了一步

遗憾的是，聚类会花费很长的时间，运行如下代码清单中的代码时你就会意识到这一点，它引入了TreeClusteringRecommender来实现这一思想。

代码清单4-10　创建一个基于聚类的推荐程序

```
UserSimilarity similarity = new LogLikelihoodSimilarity(model);
ClusterSimilarity clusterSimilarity =
    new FarthestNeighborClusterSimilarity(similarity);
return new TreeClusteringRecommender(model, clusterSimilarity, 10);
```

用户间的相似性通常由UserSimilarity的实现来定义。用户簇之间的相似性则由ClusterSimilarity的实现来定义。目前，有两种可用的实现：其中一个用分别取自两个簇的两个最相似用户之间的相似度作为聚类相似度；另一个则用分别取自两个簇的两个最不相似用户之间的相似度作为聚类相似度。

两种方式都是合理的，但也都会遇到同样的问题，即一个簇边界上的某个离群点会破坏簇的相似性。最相似用户规则对应的实现为NearestNeighborClusterSimilarity，两个成员之间平均距离很远的簇，可能因为边界接近而被它认为是相近的。最不相似用户规则对应的实现为FarthestNeighborClusterSimilarity（见代码清单4-10），它则可能因为存在两个相隔很远的离群点，而认为两个实际上非常接近的簇距离很大。

尽管当前Mahout中没有实现，但还有第三种可行的方法，即基于两个簇中心（或均值）的距离定义簇的相似性。

4.7　对比其他推荐算法

本书前面提到过，基于内容的推荐是一种广泛并经常被提及的推荐方法，它考虑了物品的上下文或属性。基于这个原因，它类似但又不同于协同过滤方法，后者仅仅基于用户与物品之间的关联，并将物品看成是没有属性的黑盒子。尽管Mahout基本上不实现基于内容的方法，但它提供了一些在推荐中使用物品属性的可能性。

4.7.1　为Mahout引入基于内容的技术

举个例子，假设有一个在线书商囤积了一些书的多个版本。这个书商可能需要为其顾客推荐图书。当然，这里的物品就是书，我们很自然会用ISBN（一个唯一的产品标识）来代表一本书。但是对于流行的大众化图书，例如《简·爱》，可能有不同的出版商出版文字相同的不同印刷品，这些印刷品的ISBN是不同的。看起来基于它们的文字来推荐书籍，要比基于不同的版本更自然一些——你应该不太在意自己读的是“《简·爱》”，还是“ACME在1993年出版的平装《简·爱》”吧？可能将书本身（它的文字）看做一个物品，并一视同仁地推荐这本书的所有版本，要比将不同版本的《简·爱》看做不同的物品来推荐更有用。从某种意义上讲，这就属于基于内容的推荐了。把一本书中的文字（这是基于内容推荐的显著特征）作为协同过滤的推荐物品，然后应用Mahout中的协同过滤技术，这个书商就完成了一种基于内容的推荐。

回忆一下，基于物品的推荐程序需要定义给定两个物品之间的相似度。这一相似度封装在ItemSimilarity的实现中。目前为止，所有的实现都只从用户的偏好值中推导出相似度——这是经典的协同过滤。但是没有理由说相似度不能基于物品属性来计算。例如，一个电影推荐程序

可能把物品（电影）间的相似度定义为诸如流派、导演、演员、上映年份等这些电影属性的函数。在传统的基于物品的推荐程序中使用这种实现也算是一个基于内容进行推荐的例子。

4.7.2 深入理解基于内容的推荐算法

更进一步，我们可以把基于内容的推荐看做是更具一般性的协同过滤。在协同过滤中，我们基于偏好值来进行计算，而偏好值就是用户与物品之间的关联。但“驱动”这种用户与物品之间关联的又是什么呢？就是用户对特定的物品属性有着某种隐含的偏好，结果完全反映在用户对特定物品的偏好值上。例如，如果朋友告诉你她喜欢*Led Zeppelin I*、*Led Zeppelin II*和*Led Zeppelin III*这些专辑，你可以比较有把握地猜测她实际上是对这些物品的一个属性表达了偏好：Led Zeppelin乐队。通过发掘这些关联并发觉物品的属性，我们就有可能基于这些对用户–物品关联更细致的了解来构建推荐引擎。

这些技术与搜索和文档检索技术类似：基于用户与属性之间的关联和物品的属性来判断一个用户可能会喜欢什么物品，跟基于查询词项和文档词项的出现来确定检索结果是类似的。尽管Mahout的推荐程序尚未包含这些技术，但在将来的版本中这会是一个很自然的改进方向。

4.8 对比基于模型的推荐算法

基于模型的推荐是Mahout未来的又一发展方向。这类技术试图基于已有的偏好值为用户偏好建立某种模型，然后推测新的偏好值。这些技术通常可被认为是更广义的协同过滤，因为它们仅仅基于用户偏好值来进行推荐。

模型可以是用户偏好值的概率图，例如可以是贝叶斯网络的形式。随后算法会试图基于现有的用户偏好值来判断用户喜欢一个新物品的概率，并在此基础上对推荐结果进行排序。

关联规则学习可以用于类似的推荐场合。通过从数据中学习诸如“如果用户喜欢物品X和物品Y，那他们也会喜欢物品Z”之类的规则，并评估这些规则的可信度，一个推荐程序就可以得到新的，且可能最受欢迎的物品的集合。

基于聚类的推荐程序也算是一种基于模型的推荐程序。聚类所得到的簇就表示一个模型，该模型描述用户如何成组，进而描述他们的偏好值缘何具有相似性。如果只从这一点上看，Mahout支持基于模型的推荐程序。但在本书写作期间，Mahout中这一部分内容仍然处于紧锣密鼓的开发之中。

4.9 小结

本章，我们深入探讨了Mahout中核心的推荐算法。

通过现实世界中的示例，我们解释了一般意义上的基于用户的推荐算法。接下来，我们了解了这个算法在Mahout中的实现，即`GenericUserBasedRecommender`。这一通用方法有很多可以定制的地方，例如用户相似度和用户邻域的定义。

我们了解了经典的用户相似性度量，即基于皮尔逊相关系数的相似性度量，提到了这种方法

存在的一些问题，并讨论了一些解决方法，例如加权。另外，我们也介绍了基于欧氏距离、斯皮尔曼相关系数、谷本系数以及对数似然比的相似性度量。

接下来我们介绍了另一类典型的推荐技术，即基于物品的推荐，其对应实现为GenericItem-BasedRecommender。它用到了前面介绍基于用户的推荐程序时提到的一些概念，例如皮尔逊相关系数。

然后，我们介绍了slope-one推荐算法，它是一个独特而又相对简单的方法，基于物品之间偏好值的平均差异进行推荐。得到这些平均差异值需要预先进行大量计算，并且要花费大量空间来存储它们，所以我们介绍了如何同时在内存和数据库中存储它们。

最后，我们简要介绍了当前框架中一些较新的、试验性质的实现，包括基于奇异值分解、线性插值以及聚类方法的实现。由于目前仍在开发之中，这些方法的应用可能仅限于小数据集上的实验使用或用于学术研究。

各种实现的关键参数和特性汇总在表4-8中。

表4-8　Mahout中现有推荐程序实现的汇总，包括选择一种实现时需要考虑的关键输入参数及关键特性

实　　现	关键参数	关键特性
GenericUserBasedRecommender	• 用户相似性度量 • 邻域定义及其大小	• 传统的实现 • 用户数较少时相对较快
GenericItemBasedRecommender	• 物品相似性度量	• 物品数相对较少时较快 • 存在物品相似性的外部定义时比较有效
SlopeOneRecommender	• 差异值存储方式	• 在运行时进行推荐和更新数据都很快 • 需要预先进行大量计算 • 适合物品数相对很少的情况
SVDRecommender	• 特征数量	• 效果很好 • 需要预先进行大量计算
KnnItemBasedRecommender	• 中值个数（k） • 物品相似性度量 • 邻域大小	• 物品数较少时效果很好
TreeClusteringRecommender	• 聚类个数 • 聚类相似性定义 • 用户相似性度量	• 在运行时推荐很快 • 需要预先进行大量计算 • 用户数较少时效果很好

我们已经介绍了Mahout对推荐引擎的支持情况，现在可以从实际应用的角度出发，去尝试更大规模的真实数据集了。你可能会奇怪：为什么到目前为止很少提及Hadoop？Hadoop是一个强有力的工具，当需要使用很多机器处理大数据集时，它是必不可少的。但它也存在缺陷：它所处理的是资源密集型的庞大计算，因此这些计算的完成时间以小时计，而不是毫秒。我们将在第一部分的最后一章讨论Hadoop。

在下一章中，我们首先将基于Mahout创建一个运行在一台机器上的、可用于生产环境的推荐引擎，它可以在不到1 s的时间内对推荐请求作出响应，并能够快速更新信息。

第5章

让推荐程序实用化

本章内容

- ❑ 分析来自真实约会网站的数据
- ❑ 设计并调优一个推荐引擎
- ❑ 在生产环境中部署一个基于Web的推荐服务

迄今为止，本书介绍了Apache Mahout所提供的推荐算法及其变种，并讨论了如何评估一个推荐程序的精度和性能。下一步是把它们都应用在真实数据集上，从零开始创建一个高效的推荐引擎。我们将依赖某个约会网站的数据创建一个推荐引擎，并把它变成一个可部署在生产环境中的Web服务。

没有一个标准的方法指导我们在给定的数据和问题域上构建推荐程序。但至少，数据必须能够表达用户和物品之间的关联，其中这里的用户和物品可以是很多对象。为推荐算法设定输入数据的过程通常是与特定问题相关的。而你如何找到最优的推荐引擎来适应输入数据也同样与场景有关。这不可避免地涉及在真实数据上进行实地的发掘、实验和评估。

本章给出一个过程间紧密衔接的示例，据此讲述如何在数据集上使用Mahout开发推荐系统。首先选取一个方法，然后收集数据、评估结果，再多次重复这个过程。其中有许多方法并不会被用到，但它们也同样重要。这种暴力的方法是有道理的，因为它可相对轻松地评估Mahout中的一个方法。另外，就像在其他问题域中一样，仅仅观察数据并不总能搞清楚什么是正确的方法。

5.1 分析来自约会网站的样本数据

下面使用一个新的数据集，它来自捷克的一个约会网站——Líbímseti（http://libimseti.cz/）。该网站的用户可以对其他用户的档案进行评分，分值从1到10不等。分值为1代表NELÍBÍ（即不喜欢），分值为10代表LÍBÍ（即喜欢）。网站展示的档案可以让该网站的用户对已建档用户的气质、形象以及可约会性做一些评价。大量的这类数据被匿名化，可供研究使用，并已由Vaclav Petricek发布（http://www.occamslab.com/petricek/data/）[①]。你需要从该网站上下载完整的数据副本

---

① 另参见Lukas Brozovsky和Vaclav Petricek的“Recommender System for Online Dating Service”，网址为http://www.occamslab.com/petricek/papers/dating/brozovsky07recommender.pdf。

(http://www.occamslab.com/petricek/data/libimseti-complete.zip)，以便使用本章中的示例。[①]

该数据集有17 359 346份评分，几乎是我们之前所用GroupLens数据集的两倍。它包含用户对物品的明确评分，这里的物品为其他人的用户档案。这意味着建立在该数据上的推荐系统是将人推荐给人。这提醒我们从更宽广的视角来理解推荐程序，而不只是局限在书和DVD这样的推荐对象上。

生成推荐程序首先要做的事情是分析所要用到的数据，并开始琢磨什么样的推荐算法才是适合的。你下载的ratings.dat文件有257 MB，是一个简单的以逗号分界的文件，包含用户ID、档案ID和评分，每行代表一个用户对另一个用户的档案的一次评分。数据被有意地做了模糊处理，因此用户ID并非网站上的真实用户ID。而档案就是其他用户的档案，数据则代表对其他用户的评分。你可能会认为这里用户ID和档案ID是对等的，比如用户ID 1和档案ID 1是同一个用户，但因为做了匿名处理，实际上并非如此。

在数据中存在135 359个独立用户，总共评价了168 791个独立的用户档案。因为用户和物品的个数大致相同，基于用户和基于物品的推荐都不会明显更优。假设档案数比用户数大得多，则基于物品的推荐程序会更慢。这里可以用slope-one，即便它的内存需求会随着物品个数快速增加，但可以对此进行限制。

这个数据集经过了预处理：剔除了生成评分个数不到20个的用户。而且，其中排除了几乎对每个档案都给出相同分值的用户，因为这可能是垃圾信息或不严肃的评分。如果来自用户的数据是他们真心做出的评分，想必相比于不那么投入的用户做的评分，他们的输入数据有用且无噪声的。

输入数据的格式直接可以用于Mahout的`FileDataModel`。即用户和档案ID是数字，文件按字段依次以逗号分隔：用户ID，物品ID和偏好值。

下载的压缩包中还有另一个有趣的数据集gender.dat：大部分档案的用户性别。不是所有的档案都有性别信息，在gender.dat中，有些行以U结尾，表示性别未知。在数据集中给出的不是用户的性别——而是档案的性别——但是我们还是多知道了一些物品的信息。当被推荐时，男性档案之间会比男性档案和女性档案之间表现得更像，反之亦然。如果某个用户的大多数或全部评分都面向男性档案，就有理由相信该用户对与男性档案约会的意愿远比女性档案更强。这个信息会成为物品之间相似性评估的基础。

这并不是一个完美的假设。这里并不转向性别这个敏感的话题，我们注意到网站的一些用户可能会乐此不疲地评论别人的档案，即使他们根本不会去约会属于该档案性别的人。一些用户还可能拥有对两种性别的浪漫情节。事实上，在ratings.dat中最开始的两个评分来自于同一个用户，显然面向的是两种不同性别的档案。

在这种约会网站的推荐引擎中考虑性别是非常重要的；如果把一个女性推荐给一个仅对男性感兴趣的用户，它必然会是一个很糟糕的推荐，而且会有些冒犯用户。做这个限制是非常重要的，但对于我们在Mahout中见到的标准推荐算法而言，这种限制并不是完全契合的。本章后续各节将检视如何作为一个过滤器和一种相似性度量方法将之插入推荐算法中。

---

① 该网站及数据发布者均与本书无关。

5.2 找到一个有效的推荐程序

为了新建一个推荐引擎来处理Líbímseti数据，你需要从Mahout中挑选一个推荐程序。这个推荐程序应该既快又好。两相权衡，最好先关注推荐的质量，再考虑性能。毕竟，飞快地生成一个很差的推荐有什么用呢？

观察数据不可能推测出正确的实现，需要做一些尝试性的测试。准备好第2章的推荐程序评估框架之后，就可以搜集各种实现在这个数据集上的表现了。

5.2.1 基于用户的推荐程序

自然先从基于用户的推荐程序开始。在Mahout中可以选择多种不同的相似性度量和邻域定义。为了了解哪些好用哪些不好用，你可以尝试许多种组合。在我们的测试环境中的一些实验结果参见表5-1、表5-2和图5-1、图5-2。

表5-1 基于一种相似性度量和用户的*n*个最近邻评估基于用户的推荐程序，所得到的估计和实际偏好值之间的平均绝对差值

相似性度量标准	*n*=1	*n*=2	*n*=4	*n*=8	*n*=16	*n*=32	*n*=64	*n*=128
欧氏距离	1.17	**1.12**	1.23	1.25	1.25	1.33	1.48	1.43
皮尔逊相关系数	1.30	1.19	1.27	1.30	1.26	1.35	1.38	1.47
对数似然	1.33	1.38	1.33	1.35	1.33	1.29	1.33	1.49
谷本系数	1.32	1.33	1.43	1.32	1.30	1.39	1.37	1.41

表5-2 基于一种相似性度量和用户的基于阈值的最近邻评估基于用户的推荐程序，所得到的估计和实际偏好值之间的平均绝对差值。一些值为“not a number”（未定义），由Java的NaN符号表示

相似性度量标准	*t*=0.95	*t*=0.9	*t*=0.85	*t*=0.8	*t*=0.75	*t*=0.7
欧氏距离	**1.33**	1.37	1.39	1.43	1.41	1.47
皮尔逊相关系数	1.47	1.4	1.42	1.4	1.38	1.37
对数似然	1.37	1.46	1.56	1.52	1.51	1.43
谷本系数	NaN	NaN	NaN	NaN	NaN	NaN

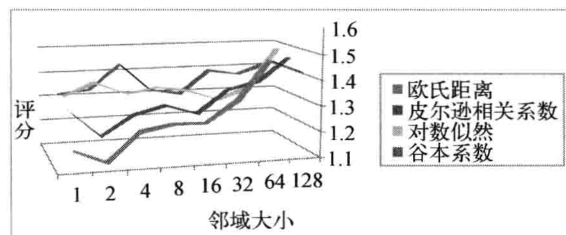


图5-1 表5-1中值的可视化

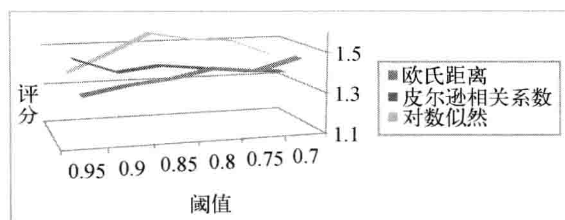


图5-2　表5-2中值的可视化

上面的结果还不错。这些推荐程序所估计的用户偏好平均偏差在1.12~1.56之间，而取值范围为1~10。

这里可以得到一些趋势，尽管有些独立的评估结果与此趋势不同。看起来欧氏距离相似性度量似乎比皮尔逊相关系数略好，虽然它们的结果非常相似。此外，小的邻域比大的好，邻域为两人时所得评估结果最佳。也许用户的偏好的确相当个性化，在计算中引入太多其他的人不会有什么帮助。

如何解释基于谷本系数的相似性度量得到的NaN（非数字）结果呢？将它列出是为了凸显该度量方式的特殊性。虽然所有的相似性度量都返回一个介于−1~1之间的值，而且值越高意味着相似性越大，但每个相似性度量下的值并不代表相同的含义。这是普遍的真理，而非Mahout故意为之。例如，在基于皮尔森相关系数的度量中0.5表示中等相似度，然而对于谷本系数，0.5意味着两个用户非常相似：他们所共知的物品占了全部物品的一半。

即使0.7~0.95的阈值对于其他度量是合理的，但对于基于谷本系数的相似性度量而言，它们却相当大。对于每次测试，这个门槛都设得太高了，导致无法建立用户的邻域！这里，也许从0.4向下测试阈值更为有用。事实上，设阈值为0.3，最佳的评估分值接近1.2。

类似地，虽然对于*n*个最近邻构成的邻域数据显然存在一个最优的*n*值，但是对于基于阈值的用户邻域，根本不存在一个同样的最优值。例如，基于欧氏距离的相似性度量似乎当阈值增加时会取得更好的结果。对于在邻域中的那些最有价值用户，它们基于欧氏距离的相似度大体上应该超过0.95。那么取0.99会怎样？或者0.999？评估结果反而下降为1.35左右；不是很糟，但显然并非最好的推荐程序。

你可以继续寻找更好的配置参数。但是，这里我们在Mahout中选用的最佳方案为：

- 基于用户的推荐程序；
- 欧氏距离相似性度量；
- 两个最近邻的邻域。

5.2.2　基于物品的推荐程序

基于物品的推荐程序只需要选择一种物品的相似性度量方法。最直接的方法是把每个相似性度量都尝试一遍，看哪种最好用。这样做之后的输出结果汇总在表5-3中。

表5-3 基于多种不同的相似性度量评估一个基于物品的推荐程序，得到估计和实际偏好值之间的平均绝对差值

相似性度量标准	评　分
欧氏距离	2.36
皮尔逊相关系数	2.32
对数似然	2.38
谷本系数	2.40

评分显然下降很多；平均误差，即估计值和实际值的平均差值，翻了大概两倍，具体值超过了2。对于上述数据，基于物品的推荐方法不太有效。为什么呢？之前，算法以基于用户的方式计算用户间的相似度，以用户对其他用户的档案评分为依据。现在，它计算的是用户档案之间的相似度，依据的是其他用户对档案的评分。也许这样做的意义不大：评分所表达的信息，更多是关于评价者（rater）的，而不是关于被评价档案（rated profile）的。

无论如何，从结果中可以清晰地看到：在这里，基于物品的推荐并非最佳选择。

5.2.3 slope-one推荐程序

回顾一下，slope-one推荐程序在数据模型中的大多数物品对之间求得一个差值。这里有168 791个物品（档案），就意味着潜在存储了280亿个差值——它过于庞大而无法存入内存。或许可以在数据库中存储这些差值，但这会极大地降低性能。

幸运的是，还有另一种选择，即通过框架将存储的差值个数限制在大约一千万个，如代码清单5-1所示。框架会试图选择最有用的差值来保存。这里所指的最有用的差值是指那些最经常一起出现的物品对之间的差值。例如，如果物品A和B出现在几百个用户的偏好值中，在它们的偏好值中的平均差值会非常大，而且是有用的。如果A和B仅一起出现在一个用户的偏好值中，它更多的是一个巧合的数据，而不值得去存储。

代码清单5-1　通过MemoryDiffStorage限制内存占有量

```
DiffStorage diffStorage = new MemoryDiffStorage(
    model, Weighting.WEIGHTED, 10000000L);
return new SlopeOneRecommender(
    model, Weighting.WEIGHTED, Weighting.WEIGHTED, diffStorage);
```

通过检查Mahout输出的日志可知，这个方法所占用的内存量大约为1.5 GB。你还会注意到Slope-one的速度，在我们做测试的工作站上，平均的推荐时间小于10 ms，而其他算法则需要大约200 ms。

评估结果为1.41左右。这还不错，但并没有好到基于用户的推荐的水平。似乎对于这个特定的数据集，没有必要花力气使用Slope-one。

5.2.4 评估查准率和查全率

之前的案例试用了基于对数似然比的相似性度量和基于谷本系数的相似性度量，它们都没有使

用用户的偏好值。但是，这些案例却无法对完全忽略偏好值的推荐程序进行评价。无法以相同的方式评估这些推荐程序——没有估计的偏好值可以去和实际值做比较，因为根本就没有偏好值。

使用RecommenderIRStatsEvaluator可以比较这些推荐程序相对于当前最佳方案（基于用户的推荐程序，采用欧氏距离测度且邻域为2）的查准率和查全率。该评估程序显示了推荐结果为10时的查准率和查全率（就是说，查准率和查全率由前10个推荐结果来评判），它们分别约为3.6%和5%。值似乎比较低：这个推荐程序基本不会推荐用户自己评分高的那些档案。不过在该场景下，这未必是坏事。可以想象用户在约会网站上所能找到的、可评为10分的完美档案可能有很多，但用户遇到和评价过的大概只占其中很少一部分。最好是由推荐程序来提出更好的建议，而不是仅给出用户评过分的那些档案！的确如此，这正是推荐程序所应传达的，那些用户评分很高的档案并不一定是他们最喜欢的，除非他们真的看过每一个已有的档案。

当然，另一种解释是推荐程序的执行有问题。但是，鉴于这个推荐程序可以很好地估计出偏好值，所估计的分值在10分制中只有大约1分左右的误差。因此，上一段的解释应该是对的。

如果我们忽略评分数据进行推荐，需使用Mahout的GenericBooleanPrefDataModel、GenericBooleanPrefUserBasedRecommender和一个像LogLikelihoodSimilarity这样的合适的相似性度量，此时就会出现一件有趣的现象。在这个案例中，查准率和查全率增长到22%以上。使用TanimotoCoefficientSimilarity也会看到类似的结果。这些结果表面上看起来变好了，它们意味着这种（基于布尔型偏好的）推荐引擎更擅长向用户推荐他们可能已经见过的那些档案。如果有确凿的证据显示用户确实看过了大部分档案，他们实际评为高分的档案会是指引我们找到正确答案的一个明确信号。但是，对于有几十个档案的约会网站，这并不适用。

在其他场景中，获得高的查准率和查全率也许是很重要的，但在这里似乎并非如此。至于我们，仍将继续使用之前基于用户的推荐程序（基于欧氏距离相似度和大小为100的邻域），而不会改用其他的推荐程序。

5.2.5　评估性能

评估我们所选择的推荐程序的运行性能非常重要。加为要支持实时查询，所以实现一个要在几分钟时间内完成推荐的推荐程序意义不大。

与以前一样，可以使用LoadEvaluator类评估每个推荐所用的时间。我们在该数据集上运行这个推荐程序，采用如下的标志类参数：-server -d64 -Xmx2048m -XX:+UseParallelGC -XX:+UseParallelOldGC。我们会发现在测试机上平均每次推荐会用218 ms。这个应用在运行时仅占用1 GB左右的堆空间。这些测试结果是否可被接受，这依赖于应用的需求和可用的硬件资源。对于许多应用而言，这些测试数据应该还是符合要求的。

至此，我们只是在手头的数据集上应用了标准的Mahout推荐程序。我们没有做任何定制。但是，为一个特定的数据集或网站生成最优的推荐系统不可避免地需要利用所有已知的信息。进而，这需要对Mahout中的那些标准实现做些定制和特殊处理，来利用关于手头问题的特殊属性。在下一节，我们将对现有的Mahout实现做一些扩展，利用约会数据的特定属性来提高推荐的质量。

5.3 引入特定域的信息

至今推荐程序没有利用任何特定的背景知识。具体来讲，它还没有利用这样的事实，即评分发生在人和人之间。推荐程序完全使用用户档案评分，就好像针对图书、汽车或水果的评分。但是，通常可以通过数据中额外的信息来改善推荐质量。

本节中，我们将寻求引入该数据集中一个尚未使用的重要信息：性别（gender）。我们基于性别定制了一个ItemSimilarity度量，并力求避免推荐性别不当的用户。

5.3.1 采用一个定制的物品相似性度量

因为已经给定了许多档案的性别，你可以仅仅基于性别为档案对建立一个简单的相似性度量。因为档案就是这里的物品，所以它在框架中应该是ItemSimilarity。

例如，认为两个男性或者两个女性档案非常相似，并设置它们的相似度为1.0。假定男性和女性档案之间的相似度为-1.0。最后，一对档案中一个或两个的性别未知，则设两者的相似度为0.0。

这个想法很简单，也许过度简单了。它应该会非常快，但是会在度量计算中丢失与评分相关的所有信息。对于该实验，我们采用一个基于物品的推荐程序，如下所示。

代码清单5-2 一个基于性别的物品相似性度量

```
public class GenderItemSimilarity implements ItemSimilarity {

  private final FastIDSet men;
  private final FastIDSet women;

  public GenderItemSimilarity(FastIDSet men, FastIDSet women) {
    this.men = men;
    this.women = women;
  }

  public double itemSimilarity(long profileID1, long profileID2) {
    Boolean profile1IsMan = isMan(profileID1);
    if (profile1IsMan == null) {
      return 0.0;
    }
    Boolean profile2IsMan = isMan(profileID2);
    if (profile2IsMan == null) {
      return 0.0;
    }
    return profile1IsMan == profile2IsMan ? 1.0 : -1.0;
  }

  public double[] itemSimilarities(long itemID1, long[] itemID2s) {
    double[] result = new double[itemID2s.length];
    for (int i = 0; i < itemID2s.length; i++) {
      result[i] = itemSimilarity(itemID1, itemID2s[i]);
    }
    return result;
```

```
    }
  ...
    private Boolean isMan(long profileID) {
      if (men.contains(profileID)) {
        return Boolean.TRUE;
      }
      if (women.contains(profileID)) {
        return Boolean.FALSE;
      }
      return null;
    }

    public void refresh(Collection<Refreshable> alreadyRefreshed) {
      // 什么也不做
    }

  }
```

和前面的案例一样，这个ItemSimilarity度量可以与标准的GenericItemBased-Recommender一起使用，我们可以评估这个推荐程序的精度。这个概念很有趣，但是这里得到的结果为2.35，并不比其他的度量更好。如果可以获得更多的信息，比如每个档案所表达的兴趣和爱好，就可以作出一个更有意义的相似性度量，也许会得到更好的结果。

但是，这个示例给出了基于物品的推荐程序的主要优点：它提供了一种结合物品自身信息的手段，这在推荐程序中很常见。从评估的结果看，你也许还会注意到这种基于简捷计算相似性度量的推荐程序速度有多快；在我们的测试节点上，生成推荐结果的时间平均为15 ms左右。

5.3.2　基于内容进行推荐

刚刚出现的一个要点，可能会被你忽略掉：上一节讲的是一个基于内容进行推荐的案例。它对物品相似性的定义不是基于用户的偏好，而是基于物品自身的属性。如前所述，Mahout并不提供基于内容推荐的实现，但是却支持扩展并提供了API，允许你在框架中写代码来部署这种实现。

对于仅基于用户偏好的协同过滤方法而言，这种实现是一个很好的补充。你可以很好地引入自己对物品（物品在此处是指人）的知识，来强化你手里的用户偏好数据，进而有望生成更好的推荐结果。

遗憾的是，前面的物品相似性度量针对的是手头的特定问题域。这种度量对其他问题域毫无帮助：推荐食品、电影或者旅行目的地等。这就是它没有成为框架一部分的原因。但是只要你拥有特定问题域的知识，它就是一种可行和有力的方法，它在描述物品相关关系时比用户偏好更为有效。

5.3.3　利用**IDRescorer**修改推荐结果

你可能已经观察到Recommender.recommend()方法中有一个类型为IDRescorer的用final修饰的可选参数，你可以不调用recommend(long userID, int howMany)，而调用recommend(long userID, int howMany, IDRescorer rescorer)。这个接口的实现多次

出现在Mahout推荐程序的相关API中。它可以根据某种逻辑将推荐引擎中的某些值修改为其他值，也可以在某个过程中将一个实体排除出去。例如，IDRescorer可以任意地修改Recommender对一个物品的估计偏好值。它还可以将一个物品从考虑范围内彻底去除掉。

假如你正在为一个电子商务网站的用户推荐图书。如果用户正在浏览的是悬疑小说，那么在向该用户推荐图书时，你可能希望将所有悬疑小说的估计偏好值都提高一些。你或许还希望确保不去推荐那些缺货的书。IDRescorer可以帮助你做到这一点。下面的代码清单显示了一个IDRescorer的实现，它根据这个虚构的书商来虚构出一些实现类，如Genre（流派），并以此封装了这一逻辑。

5

代码清单5-3 示例IDRescorer忽略缺货图书并提高一个流派的估计值

```
public class GenreRescorer implements IDRescorer {

  private final Genre currentGenre;

  public GenreRescorer(Genre currentGenre) {
    this.currentGenre = currentGenre;
  }

  public double rescore(long itemID, double originalScore) {
    Book book = BookManager.lookupBook(itemID);        <-- 假设BookManager存在
    if (book.getGenre().equals(currentGenre)) {
      return originalScore * 1.2;                      <-- 将估计值提高20%
    }
    return originalScore;                              <-- 其他保持原状
  }

  public boolean isFiltered(long itemID) {
    Book book = BookManager.lookupBook(itemID);
    return book.isOutOfStock();                        <-- 过滤缺货的图书
  }
}
```

rescore()方法将悬疑小说的估计偏好值提高了。isFiltered()方法显示了IDRescorer的另一个用途：它确保了缺货的图书不会被推荐。

这只是一个示例，和我们的推荐网站没有关系。让我们把这个思想应用到可用的附加数据上：性别。

5.3.4 在IDRescorer中引入性别

对于在乎性别的用户，IDRescorer能够对物品或用户档案进行过滤。首先，可以通过检查已经评价过的档案的性别，来猜测该用户所偏好的性别。然后，就可以滤除与之性别相反的档案，如下所示。

代码清单5-4 基于性别的IDRescorer实现

```
public class GenderRescorer implements IDRescorer {

  private final FastIDSet men;
```

```
private final FastIDSet women;
private final FastIDSet usersRateMoreMen;
private final FastIDSet usersRateLessMen;
private final boolean filterMen;

public GenderRescorer(FastIDSet men,
                      FastIDSet women,
                      FastIDSet usersRateMoreMen,
                      FastIDSet usersRateLessMen,
                      long userID, DataModel model)
    throws TasteException {
  this.men = men;
  this.women = women;
  this.usersRateMoreMen = usersRateMoreMen;
  this.usersRateLessMen = usersRateLessMen;
  this.filterMen = ratesMoreMen(userID, model);
}

public static FastIDSet[] parseMenWomen(File genderFile)
    throws IOException {
  FastIDSet men = new FastIDSet(50000);
  FastIDSet women = new FastIDSet(50000);
  for (String line : new FileLineIterable(genderFile)) {
    int comma = line.indexOf(',');
    char gender = line.charAt(comma + 1);
    if (gender == 'U') {
      continue;
    }
    long profileID = Long.parseLong(line.substring(0, comma));
    if (gender == 'M') {
      men.add(profileID);
    } else {
      women.add(profileID);
    }
  }
  men.rehash();
  women.rehash();
  return new FastIDSet[] { men, women };
}

private boolean ratesMoreMen(long userID, DataModel model)
    throws TasteException {
  if (usersRateMoreMen.contains(userID)) {
    return true;
  }
  if (usersRateLessMen.contains(userID)) {
    return false;
  }
  PreferenceArray prefs = model.getPreferencesFromUser(userID);
  int menCount = 0;
  int womenCount = 0;
  for (int i = 0; i < prefs.length(); i++) {
    long profileID = prefs.get(i).getItemID();
    if (men.contains(profileID)) {
```

缓存更多对男性评分的用户

之后分别被调用

快速访问的重新优化

```
        menCount++;
      } else if (women.contains(profileID)) {
        womenCount++;
      }
    }
    boolean ratesMoreMen = menCount > womenCount;      <-- 对男性评分的用户可能更喜欢男性档案
    if (ratesMoreMen) {
      usersRateMoreMen.add(userID);
    } else {
      usersRateLessMen.add(userID);
    }
    return ratesMoreMen;
  }

  public double rescore(long profileID, double originalScore) {
    return isFiltered(profileID)
      ? Double.NaN : originalScore;      <-- 将被排除的档案赋值为NaN
  }

  public boolean isFiltered(long profileID) {
    return filterMen ? men.contains(profileID) : women.contains(profileID);
  }

}
```

这个示例代码做了几件事情。`parseMenWomen()`方法解析gender.dat并创建了两个档案ID集合——已知是男性的档案ID和已知是女性的档案ID。解析独立于任何特定的`GenderRescorer`实例之外，因为这些集合会被多次重用。`ratesMoreMen()`方法用来决定并记住一个用户是否会更多地对男性或女性档案评分。这些结果被缓存在两个额外的集合中。于是这个`GenderRescorer`的实例通过`rescore()`返回NaN，或从`isFiltered()`返回`true`，很容易在适当情况下排除掉男性或女性。

这应该对推荐的质量会帮助，但不会很大。大体上，对男性档案评分的女性已经被推荐了男性档案，因为她们最类似于其他对男性档案评分的女性。这个机制通过从结果中排除女性档案来确保这一点。这会让推荐程序甚至不去试图估计这些女性对女性档案的偏好，因为这种估计纯粹是臆测，很可能会出错。当然，这个`IDRescorer`的效果受限于已有数据的质量：只有大约一半档案的性别是已知的。

5.3.5 封装一个定制的推荐程序

对于我们而言，将现有的推荐引擎和这个新`IDRescorer`封装在一个实现中是很有用的。这样，我们可以在5.5节部署一个完整的推荐引擎。

下面的代码清单显示了一个推荐程序实现，其中包含了前面所讲的基于用户的推荐引擎。

代码清单5-5 完成面向Líbímseti的推荐程序

```
public class LibimsetiRecommender implements Recommender {

  private final Recommender delegate;
  private final DataModel model;
```

```
  private final FastIDSet men;
  private final FastIDSet women;
  private final FastIDSet usersRateMoreMen;
  private final FastIDSet usersRateLessMen;

  public LibimsetiRecommender() throws TasteException, IOException {
    this(new FileDataModel(
      readResourceToTempFile("ratings.dat")));      <-- 在生产环境下需要
  }                                                     readResourceToTempFile()

  public LibimsetiRecommender(DataModel model)
      throws TasteException, IOException {
    UserSimilarity similarity =                          构建基于用户的推荐程序
        new EuclideanDistanceSimilarity(model);      <--
    UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(2, similarity, model);
    delegate =
        new GenericUserBasedRecommender(model, neighborhood, similarity);
    this.model = model;
    FastIDSet[] menWomen = GenderRescorer.parseMenWomen(
        readResourceToTempFile("gender.dat"));
    men = menWomen[0];
    women = menWomen[1];
    usersRateMoreMen = new FastIDSet(50000);
    usersRateLessMen = new FastIDSet(50000);
  }
  public List<RecommendedItem> recommend(long userID, int howMany)
      throws TasteException {
    IDRescorer rescorer = new GenderRescorer(
        men, women, userID, usersRateMoreMen, usersRateLessMen,
        userID, model);                                <-- 在所有推荐上使用
    return delegate.recommend(userID, howMany, rescorer);  GenderRescorer
  }

  public List<RecommendedItem> recommend(long userID,
                                         int howMany,
                                         IDRescorer rescorer)
      throws TasteException {
    return delegate.recommend(userID, howMany, rescorer);
  }

  public float estimatePreference(long userID, long itemID)
      throws TasteException {
    IDRescorer rescorer = new GenderRescorer(
        men, women, userID, usersRateMoreMen, usersRateLessMen,
        userID, model);                                      <--
    return (float) rescorer.rescore(                            重算估计偏好
        itemID, delegate.estimatePreference(userID, itemID));
  }

  public void setPreference(long userID, long itemID, float value)
      throws TasteException {
    delegate.setPreference(userID, itemID, value);      <--
  }                                                        委托给底层的推荐程序

  public void removePreference(long userID, long itemID)
```

```
        throws TasteException {
      delegate.removePreference(userID, itemID);
    }

    public DataModel getDataModel() {
      return delegate.getDataModel();
    }

    public void refresh(Collection<Refreshable> alreadyRefreshed) {
      delegate.refresh(alreadyRefreshed);
    }

}
```

这是一个小而完整的推荐引擎封装。如果对它进行评估，结果大约为1.18（这实际上是恒定不变的），最好改用这种机制来避免一些有严重偏差的推荐。运行时间会增加到500 ms左右——重算引入了大量的开销。对于我们的目标而言，这一折中是我们可以接受的，而LibimsetiRecommender正是这个约会网站的最终实现。

5.4 为匿名用户做推荐

当你创建一个实用化的推荐程序时会马上遇到另一个常见问题，即如何处理那些尚未注册的用户。例如，如何处理在一个电子商务网站上浏览商品的新用户？对于网站而言，这个匿名用户既没有浏览记录，也无购买历史，更不用说ID号了？为这样的用户进行推荐的问题在于无数据可用，故而称为*冷启动问题*。但能够为这样的用户推荐商品也是有用的。

一种极端的方法是不做个性化的推荐。就是说，当面对一个新用户时，提供一个普通的产品预定义列表作为推荐。这样做很简单，并且通常比不做强，但不在我们的选择之列。Mahout所关注的是个性化推荐。

另一个极端做法是，网站将这种匿名用户在第一次访问时就升级为真实用户，赋予其一个ID，并根据网络会话来跟踪其行为。这也有效果，虽然这会潜在地导致用户数爆发式增长，但不难想象其中许多用户永远不会再来且已有的信息很少。

后者也不是我们寻找的选项。相反，我们在两种极端做法之间寻求妥协方案：生成临时用户并将所有的匿名用户当做一个用户。

5.4.1 利用PlusAnonymousUserDataModel处理临时用户

通过PlusAnonymousUserDataModel类，这个推荐程序框架提供了一个临时增加匿名用户的信息到DataModel的简单方法。这种方法将匿名用户视为真实用户，但是这仅适用于进行推荐的时候。真实的底层DataModel中不会增加这些匿名用户，也不会得知它们的存在。对于现有的DataModel而言，PlusAnonymousUserDataModel是在其上的一个封装，可以简单地做替换。

PlusAnonymousUserDataModel类在临时用户的处理上有一个特殊的地方，它每次只处理一个此类用户的偏好值。基于这个类的Recommender同样必须一次只处理一个匿名用户。

下面的代码清单展示了LibimsetiWithAnonymousRecommender，它扩展了先前的

LibimsetiRecommender，采用了一个可以向匿名用户进行推荐的方法。当然，它取偏好值作为输入，而不是用户ID。

代码清单5-6　Líbímseti上的匿名用户推荐

```
public class LibimsetiWithAnonymousRecommender
    extends LibimsetiRecommender {

  private final PlusAnonymousUserDataModel plusAnonymousModel;

  public LibimsetiWithAnonymousRecommender()
      throws TasteException, IOException {
    this(new FileDataModel(
        readResourceToTempFile("ratings.dat")));
  }

  public LibimsetiWithAnonymousRecommender(DataModel model)
      throws TasteException, IOException {
    super(new PlusAnonymousUserDataModel(model));        // 封装底层的DataModel
    plusAnonymousModel =
        (PlusAnonymousUserDataModel) getDataModel();
  }

  public synchronized List<RecommendedItem> recommend(   // 使用同步
      PreferenceArray anonymousUserPrefs, int howMany)
      throws TasteException {
    plusAnonymousModel.setTempPrefs(anonymousUserPrefs);
    List<RecommendedItem> recommendations =
        recommend(PlusAnonymousUserDataModel.TEMP_USER_ID,
                  howMany, null);                        // 设置匿名用户的ID
    plusAnonymousModel.clearTempPrefs();
    return recommendations;
  }

  public static void main(String[] args) throws Exception {
    PreferenceArray anonymousPrefs =
        new GenericUserPreferenceArray(3);               // 存储匿名用户的偏好值
    anonymousPrefs.setUserID(0,
        PlusAnonymousUserDataModel.TEMP_USER_ID);
    anonymousPrefs.setItemID(0, 123L);
    anonymousPrefs.setValue(0, 1.0f);
    anonymousPrefs.setItemID(1, 123L);
    anonymousPrefs.setValue(1, 3.0f);
    anonymousPrefs.setItemID(2, 123L);
    anonymousPrefs.setValue(2, 2.0f);
    LibimsetiWithAnonymousRecommender recommender =
        new LibimsetiWithAnonymousRecommender();
    List<RecommendedItem> recommendations =
        recommender.recommend(anonymousPrefs, 10);
    System.out.println(recommendations);
  }
// readResourceToTempFile实现被忽略
// ……
}
```

另外，这个实现和任何其他推荐程序一样工作，也可以用于向真实用户进行推荐。

5.4.2 聚合匿名用户

还可以将所有的匿名用户视为单一用户，这会简化操作。不再分别跟踪潜在的用户并单独存储他们的浏览历史，你可以将所有这样的用户看做一个大的临时用户。但这依赖于一种假设，即所有这些用户的行为是相似的。

在任何时候，这个技术都可以为匿名用户生成推荐，并非常迅速。事实上，因为结果对于所有的匿名用户都是一样的，推荐结果的集合可以被存储并周期性地重算，而不是每次请求都计算一次。在某种意义上，这种变化比根本不做个性化推荐略好，从而避免匿名用户总是看到固定的推荐。

5.5 创建一个支持 Web 访问的推荐程序

问题不仅仅是创建一个在IDE环境中运行的推荐程序，而是要把推荐程序部署在真实的产品应用中。

你也许打算在Java和Mahout中设计并测试好推荐程序，然后将其作为一个应用架构中单独的组件进行部署，而不是将它嵌入到应用的Java代码中。Web服务通常使用简单的HTTP或者SOAP之类的Web服务协议。在这里，推荐程序部署为一个Web上可见的服务，或者是一个Web容器中的独立组件，甚至作为自己的服务进程。这增加了复杂性，但会让这个服务可以由基于其他语言编写或者运行在其他机器上的应用来访问。

好在利用Mahout很容易将推荐程序捆绑成可部署的WAR（Web archive）文件。这一组件能够很好地部署在Java servlet容器中，如Tomcat（http://tomcat.apache.org/）或Resin（http://www.caucho.com/resin/）。如图5-3所示，该WAR文件封装了`Recommender`实现，通过`RecommenderServlet`这个简单的、基于servlet的HTTP服务开放出来，并基于HTTP的SOAP协议成为Apache Axis所支持的Web服务`RecommenderService`。

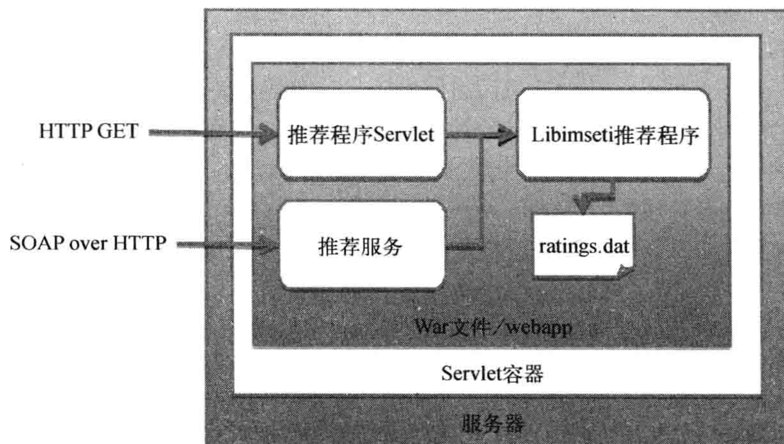


图5-3 推荐程序的自动化WAR封装及在serveletx容器中的部署

5.5.1　封装WAR文件

在部署之前，需要把编译后的代码和数据文件打包为一个JAR文件。在IDE中编译好这段代码，进入本书的示例目录，然后复制数据集里的ratings.dat和gender.dat文件到/src/main/resources目录下，再用下面的命令制作出JAR文件：

```
mvn package
```

这条命令会将结果放入target/mia-0.1.jar文件中。

然后进入Mahout发布包中的taste-web/模块目录，并从书中示例把target/mia-0.1.jar复制到lib/子目录中。再编辑recommender.properties将推荐程序命名为所要采用的名称。如果你使用的是与示例相同的Java包名，正确的值应为`mia.recommender.ch05.LibimsetiRecommender`。

现在再次执行`mvn package`。你会在target/子目录下发现一个名为mahout-taste-webapp-0.5.war的文件（文件的版本号可能会不同，当你读这本书时，Mahout应该已经发布了新的版本）。这个文件很适合立刻部署在Tomcat这样的servlet容器中。事实上，它可以直接放入Tomcat的webapps/目录而无须做进一步的修改，从而形成一个可工作的基于Web的推荐程序实例。

注意　这个.war文件的名称将成为访问服务时所用URL的一部分，你可以对它重命名使之更短，如mahout.war。

5.5.2　测试部署

还有一种办法可以让你在不安装Tomcat的情况下轻松测试包含推荐程序的Web应用，即使用Maven中内置的Jetty插件。Jetty（http://www.mortbay.org/jetty/)是一个嵌入式servlet容器，其功能与Tomcat和Resin类似。

在进行测试部署之前，你需要确保本地的Mahout安装包已经编译好并可为Maven使用。在顶级Mahout目录下执行`mvn install`，然后可以去喝杯咖啡休息一会儿，此时Maven会下载其他依赖包、编译并运行测试，大概需花费10分钟的时间。好在这件事只需做一次。

按照之前章节所述将WAR文件封装好，执行`export MAVEN_OPTS=-Xmx2048m`来确保Maven和Jetty有足够的堆空间。然后，在taste-web/目录下执行`mvn jetty:run-war`。这会在本地机器的8080端口上启动一个支持Web的推荐服务。启动会花上一些时间，因为Mahout要装载和分析数据文件。

通过Web浏览器访问http://localhost:8080/RecommenderServlet?userID=3就可以得到对用户ID 3的推荐。这正是外部应用从推荐引擎中获得推荐的方法，即向这个URL发送一个HTTP的GET请求，从返回的简单文本中解析出推荐结果：每行有一个估计的偏好值和一个物品ID，优先的推荐排在前面（如代码清单5-7所示）。

代码清单5-7 对`RecommenderServlet`发出GET请求的输出结果

```
10.0     205930
10.0     156148
8.0      162783
7.5      208304
7.5      143604
7.0      210831
7.0      173483
4.5      163100
```

要探究更正式的基于SOAP的那些Web服务API，请访问http://localhost:8080/RecommenderService.jws?wsdl查看WSDL（Web Services Definition Language）文件，它定义了这个Web服务的输入和输出。这是`Recommender` API的一个简化版本。这个Web服务描述文件可以为大多数Web服务客户端所用，以自动理解并提供对该API的访问。

如果你想直接在浏览器中访问这个服务，可以访问http://localhost:8080/RecommenderService.jws?method=recommend&useID=1&howMany=10查看这个服务基于SOAP的返回结果。返回的结果集是相同的，只是显示为一个SOAP响应（见图5-4）。

```
- <soapenv:Envelope>
  - <soapenv:Body>
    - <recommendResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      - <recommendReturn soapenc:arrayType="xsd:string[][10]" xsi:type="soapenc:Array">
        - <recommendReturn soapenc:arrayType="xsd:string[2]" xsi:type="soapenc:Array">
            <recommendReturn xsi:type="xsd:string">10.0</recommendReturn>
            <recommendReturn xsi:type="xsd:string">220429</recommendReturn>
          </recommendReturn>
        - <recommendReturn soapenc:arrayType="xsd:string[2]" xsi:type="soapenc:Array">
            <recommendReturn xsi:type="xsd:string">10.0</recommendReturn>
            <recommendReturn xsi:type="xsd:string">174211</recommendReturn>
          </recommendReturn>
```

图5-4 来自`RecommenderService`的SOAP响应在浏览器中的显示

通常此时你会理性地检查这些推荐结果。站在用户的立场上，这些推荐是不是合理的？然而，我们在这里无法知道用户是谁，以及他们喜欢什么样的档案，故而无法对结果做出直观的评判。当你自己开发推荐引擎时，事情就不再是这般场景了，那时看一下实际的推荐结果就会发现问题或找到有待改进的地方。往往需要许多轮的试验和修正，我们才能使结果成为解决问题的最佳答案。

5.6 更新和监控推荐程序

现在你已经运行起来一个基于Web的推荐服务，但是它并非一个一成不变的系统，并非运行起来就可以不管。这是一个动态的服务器，实时地接收新的信息并返回应答，并且就像所有的生产系统一样，我们自然要考虑如何更新和监控这个服务。

当然，在一个真实的推荐系统中，推荐所依据的数据总在不断改变。标准的`DataModel`实现

会自动使用来自底层数据源的最新数据，因此在上层无须做特别处理就可以让推荐引擎处理新的数据。例如，假如你通过JDBCDataModel让推荐引擎处理数据库中的数据，那么只需将新的数据更新到底层数据库表中，推荐引擎就会开始使用这些数据。

但是出于对性能的考虑，许多组件会缓存信息和中间的计算结果。这些缓存最终会被更新，但这意味着新的数据不会立即影响推荐结果。可以调用Recommender.refresh()来强制清空所有的缓存，也可以通过调用refresh方法来实现，该Web应用在基于SOAP的接口上开放了这个方法。有必要时，还可以由企业应用系统中的其他部分来调用。

需要特别关注基于文件的偏好数据，它是通过FileDataModel来访问的。在部署更新的信息时，我们可以对这个文件进行更新或覆盖，而FileDataModel会马上注意到这个更新，并重新加载这个文件。

数据文件的重新加载过程会非常慢且很耗内存，因为旧模型和新模型会同时占用内存。这时正好可以用到第3章中的更新文件（update file）。不是对这个主数据文件进行替换或更新，更有效的方式是增加表示近期更新的更新文件。更新文件就像是diffs，在与主数据文件放在同一目录中并以适当的形式命名时，它们就会被检测到，并迅速成为偏好数据的内存表示。

例如，一个应用可能每个小时都会定位所有在上一个小时或更长时间内生成的、删除的或修改的偏好数据，由此生成一个更新文件，并把它和主数据文件放在一起。如前所述，所有这些文件还应被压缩以提高效率。

可以直观地监测推荐服务的健康状态，即使这并不属于Mahout自身的范畴。只要可以通过HTTP访问来检查Web服务的健康状态，任何监测工具都可以用:访问该服务的URL并确认返回结果是否有效，这样就能够检查这个推荐服务是否还有效。同时，这样的工具可以（也应该能够）检测到应答请求的时间，并在性能突然下降时报警。一般而言，一次推荐所需的计算时间是固定的，不会有很大的变化。

5.7　小结

在本章，我们深入观察了取自Czech约会网站Líbímseti的一个真实的大型数据集。它提供了超过10万个用户对10万个以上档案的1700万个评分。我们致力于为这个网站创建一个推荐程序，让它能够为网站的用户推荐档案或者是人。

我们在这个数据集上尝试了至今所见的大多数推荐方法，并努力通过评估技术选出一个看似能够生成最佳推荐的实现：基于用户的推荐程序，它使用基于欧氏距离的相似性度量，并将邻域大小设定为2。

之后，我们尝试在推荐中引入数据集中的附加信息，即用户的性别，这是许多档案都具有的特征。我们试图基于这个数据生成一个基于物品的相似性度量。我们初识了IDRescorer接口，这是一个实用的工具，可以针对特定的问题域修正结果。我们利用IDRescorer纳入对性别的考虑，据此排除那些用户不会感兴趣的推荐，在一定程度上改进了推荐质量。

因为测得的性能是可以接受的（每次推荐约500 ms），我们便架构了一个推荐引擎的可部署

版本，并使用Mahout为它自动生成了一个支持Web的应用。我们简要地考察了如何部署以及如何通过HTTP和SOAP来访问这个组件。

最后，我们审视了如何在运行时更新推荐程序的底层数据。

我们从数据一直讨论到可用于生产环境的推荐服务，到此就算告一段落了。这个实现能够在单机上轻松自如地处理拥有1700万份评分的数据集，并能实时地生成推荐。但是，如果数据超过了单机的承载能力又如何呢？下一章，我们会考察如何基于Hadoop处理大得多的数据集。

第6章 分布式推荐

本章内容

- 分析来自维基百科的海量数据集
- 使用Hadoop和分布式算法生成推荐结果
- 将现有非分布式推荐程序改为伪分布式推荐程序

本书所用的大数据集不断增长：偏好条目从10个、10万个到1000万个不等，甚至达到1700万个。但对于推荐程序而言，这种数据集还只能算是中等规模。本章将采用维基百科上文章之间的链接所构成的海量实体，处理一个具有1.3亿个偏好的更大数据集。[①]在这个数据集中，文章既是“用户”又是物品，用于展示在非传统场景中如何通过Mahout使用推荐程序。

虽然示范所用的1.3亿个偏好的规模尚在可控范围之内，但使用现有方法在单机上处理如此大的数据集仍有难度。推荐算法亟待革新，即需要借助于Mahout所采用的MapReduce范式和Apache Hadoop实现分布式计算。

我们先审视Wikipedia数据集，来理解把一个推荐计算变为分布式计算意味着什么。考虑到与非分布式实现存在明显差异，我们先学习如何在分布式环境中设计一个简单的分布式推荐系统。你还将看到如何基于MapReduce和Hadoop在Mahout中实现它。最后，你将首次尝试运行一个完整的、基于Hadoop的推荐作业，并最后得到结果。

6.1 分析 Wikipedia 数据集

我们以考察Wikipedia数据集为起点，但之后的论述有所不同；受到数据规模问题的影响，像以前一样进行论述很困难，我们不得不先探讨计算的分布问题。

维基百科（Wikipedia，http://wikipedia.org）是一个著名的在线百科全书，其内容可由用户编辑和维护。据报道，它在2010年5月时仅英文文章就超过320万篇。Freebase Wikipedia Extraction项目（http://download.freebase.com/wex/）估计仅英文文章的大小就接近42 GB。维基百科是基于

① 看过早期草稿的读者会记得本章以前用的是Netflix Prize数据集。由于法律原因，那个数据集已经不再通过官方发布，因此不再适合用作样本数据集了。

Web的，其文章一定会链接到另外一篇文章。我们要关注的就是这些链接。我们将文章视为“用户”，把一篇文章所指向的文章视为该源文章所喜欢的物品。

幸运的是，不必下载Freebase的Wikipedia来提取并解析出所有这些链接。Henry Haselgrove已经把文章中的链接提取好并发布在http://users.on.net/~henry/home/wikipedia.htm。它进一步提取出对附属资源的链接，如文章讨论页、图像等。该数据集还采用数字化的ID号来表示文章，而不是文章的标题，这很有帮助，因为Mahout将所有用户和物品视为数字化的ID。

首先，从Haselgrove的网站上下载并提取links-simple-sorted.zip文件。这个数据集包含从5 706 070篇文章到另外3 773 865篇文章的130 160 392个链接。注意，这里没有显式的偏好或评分，只有从文章到文章的关联。这就是布尔型偏好。关联是单向的，存在一个从A到B的链接并不代表从B到A会有任何关联。物品并不会远远多于用户，反之亦然，无论基于用户还是基于物品的算法在性能上都不会更优。如果我们使用涉及相似性评估的算法，最好选择一个不依赖于偏好值的，比如`LogLikelihoodSimilarity`。

这些数据表面的意义是什么呢？可以得到什么合理的推荐结果呢？文章A到B的链接意味着B提供了与A有关的信息，通常是提供了A所引用事实或想法的背景信息。基于这些数据，推荐系统向A推荐的文章依赖于其他文章的指向，这些文章即指向推荐文章，也指向A所指向的部分文章。推荐文章可被视为A应该链接但未做链接的，它们应该也是A的读者所感兴趣的文章。有些时候，推荐会揭示有趣或偶然的关联，甚至是文章A没有暗示的。

6.1.1 挑战规模

用一个非分布式的推荐引擎来处理这些数据是很困难的。在Mahout上，该数据自身就占用了大约2 GB的堆空间，总共的堆空间大致需要2.5 GB。在一些32位的平台和JVM上，这实际上已经突破了可用堆空间大小的上限，用不了多久我们就需要用64位的机器。根据算法的不同，推荐时间可能会超过1 s，对于一个支持现代Web应用的实时推荐引擎而言，这可是一个很长的时间。

当硬件资源充足时，性能尚可接受。但如果输入增长为几十亿个偏好，堆空间需求超过32 GB，该怎么办呢？以后进一步扩展呢？有时，可以丢弃“噪声”数据来减少数据大小，以对抗规模问题。但是判断什么是噪声，这本身就是一个精度和规模的问题。

当前，如果无法处理超过一定规模的数据，使系统处理能力存在无法突破的瓶颈，可不是什么时髦的事情。我们已经能够得到海量的计算资源，这里的问题是如何把足够的计算资源集中在一起。相比于把更多小型机器攒起来，更大的机器会带来高昂的成本。这个庞大的单机还会带来单点失效问题。而且，在推荐引擎处理过程之外，很难找到一种有效的方式充分利用这台单机的昂贵处理能力。

Wikipdia链接数据集的大小代表一个实用的规模上限，它代表一个基于Mahout、非分布式、实时的推荐程序在一个硬件还不错的服务器上可以处理多大的数据——而且这个机器用当前的标准来衡量也不算太庞大。超过这个上限，就该用新的办法了。

6.1.2　分布式计算的优缺点

鉴于这些使用单台大机器的不合理性，我们的解决方案采用许多小型机器①，而不是一台大家伙。在一个机构中，可能有许多小型机器没有被充分利用，我们可以用它们额外的能力来计算推荐，如图6-1所示。而且，现在可以通过云计算提供商，如亚马逊的EC2服务（http://aws.amazon.com），来获得许多机器资源。

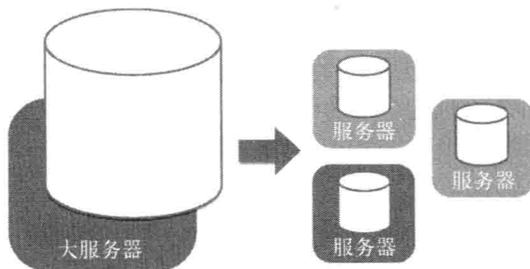


图6-1　分布式计算将一个对单台服务器过大的问题进行拆分，使之可以通过几个小服务器来处理

将推荐进行分布式计算彻底改变了推荐引擎的这个问题。迄今为止，每种计算推荐的算法在理论上都可视为基于每个偏好值的函数。为了从Wikipedia链接数据集中为某篇文章推荐一个新的链接，推荐算法就要访问所有文章到文章的链接，因为任何链接对计算都会是有用的。但是在规模很大时，无法一次访问所有数据，即便是一次访问大部分数据也是不可能的。所有我们至今已经看到的方法都失效了，至少从它们当前的状态来看是这样的。分布式推荐引擎的计算是一个全新的事物。

要澄清一点，分布式计算并不会使计算更为有效。相反，它通常会耗费更多的资源。例如，在许多小型机器之间移动数据会消耗网络资源。这种计算必然会涉及对许多中间结果的计算和存储，这样就需要大量的处理时间来做序列化、存储并在之后做反序列化。为了协调这些操作，软件会消耗不少的内存和处理资源。

需要注意，如此庞大的分布式计算必然为离线处理，而不是实时地响应用户请求。即便是很小的计算在这种形式下都需要花费好几分钟才能完成，而不是几毫秒那么短的时间。通常，在运行态下的推荐每间隔一段时间会重新计算、存储并将结果返回给用户。

但它们提供了一种途径，让推荐引擎的计算规模得以扩展，解决了一个非分布式计算因受限于单机资源而无法启动的问题。分布式计算可以利用许多机器的资源，从而将现有机器上空闲、未被使用的资源整合起来，而不是使用固定的机器资源。最终，分布式计算可以让计算过程更快完成——即便它可能会耗费更多原始的处理时间。假设一个分布式计算与非分布式计算相比用去了两倍的CPU时间。如果10个CPU来处理这个计算，它就会比仅用单个机器资源的非分布式计算快上5倍。

① “小型机器”原文为“small machine”，指的是商用计算机，如PC服务器等。——译者注

6.2 设计一个基于物品的分布式推荐算法

对于这种规模的问题，分布式生成推荐结果是合适而且必要的。之前我们已经对基于物品的推荐程序有所了解，下面会先勾勒出它的一个分布式形态。它看上去会有点儿像基于物品的非分布式推荐程序，但既然非分布式算法无法完全转换到分布式领域，它必然有所不同。

6.2.1 构建共现矩阵

算法可以通过简单的矩阵运算来完美地表达与实现。如果你只是几年前在数学课本上接触过矩阵，不要担心，只需要回忆一下矩阵的乘运算即可。这里不会用到行列式、行归约或特征值。

回顾之前介绍过的基于物品的推荐引擎，它们均依赖于一个名为`ItemSimilarity`的实现，它给出了计算任意一对物品之间相似度的方法。假设我们要计算出每个物品对之间的相似性，并将其结果导入一个巨大的矩阵。这应该是一个方阵，行和列的数目等于数据模型中的物品数。每行（以及每列）表达在一个特定物品和所有其他物品之间的相似性。事实上，把这些行和列看做向量会有助于理解。该矩阵还是沿对角线对称的，因为物品X和Y之间的相似性与物品Y和X之间的相似性是一样的，所以行X和列Y上的条目也会等于在行Y和列X上的条目。

注意 这个矩阵描述了物品之间的关联，而不涉及用户。这并非是一个用户–物品矩阵。那种矩阵不会是对称的；其行数和列数与用户和物品的数量匹配，但并不相同。

有这样一种矩阵是算法所需要的：共现矩阵（co-occurrence matrix）。它不是计算每个物品对之间的相似性，而是计算在某些用户偏好值列表中每个物品对共同出现的次数，以此来填充矩阵。例如，如果有9个用户都为物品X和Y给出了一些偏好，那么X和Y同时出现了9次。两个在任何用户偏好中均未同时出现的物品，其共现次数为0。而且，在概念上，每当用户给出对某个物品的偏好，就代表该物品与自身共生了一次，不过这个计数并没有什么用。

共现关系与相似性很像：两个物品同时出现得越多，它们越有可能相关或相似。共现矩阵的作用类似于基于物品的非分布式算法中的`ItemSimilarity`。

做简单的计数就可以生成这个矩阵。只是要注意矩阵中的条目不受偏好值的影响。这些值稍后会参与计算。表6-1给出了对一个小的偏好值样本集生成的共现矩阵，这个偏好值集合在本书中已多次用到。

表6-1 一个简单数据集中的物品共现矩阵。首行首列为行列名，不是矩阵值

	101	102	103	104	105	106	107
101	5	3	4	4	2	2	1
102	3	3	3	2	1	1	0
103	4	3	4	3	1	2	0
104	4	2	3	4	2	2	1

（续）

	101	102	103	104	105	106	107
105	2	1	1	2	2	1	1
106	2	1	2	2	1	2	0
107	1	0	0	1	1	0	1

不出意外，该矩阵是对角线对称的。因为有7个物品，所以矩阵为7 × 7的方阵。对角线上的值对算法没有意义，但是出于对完整性的考虑也被包含进来。

6.2.2　计算用户向量

在推荐程序向一个基于矩阵的分布式实现转换的下一步，我们将一个用户的偏好视为一个向量。我们之前已经讨论过基于欧氏距离的相似性度量，即将用户视为空间中的一个点，而相似性则基于它们之间的距离进行度量。

同样，在一个有*n*个物品的数据模型中，用户偏好就像一个*n*维向量，每个维度代表一个物品。用户对物品的偏好值为这个向量中的值。用户没有表达偏好的物品映射为向量中的0值。它是一个典型的稀疏矩阵，大多值为0，因为用户通常仅对一小部分物品表达偏好。

例如，在这个小型样本数据集中，用户3的偏好对应向量[2.0, 0.0, 0.0, 4.0, 4.5, 0.0, 5.0]。要生成推荐结果，每个用户都需要有这样一个向量。

6.2.3　生成推荐结果

要为用户3计算出推荐结果，只需将这个向量作为列向量，用它乘以共现矩阵，如表6-2所示。

表6-2　共现矩阵乘以用户3的偏好值向量（U3）生成推荐结果R

	101	102	103	104	105	106	107		U3		R
101	*5*	*3*	*4*	*4*	*2*	*2*	*1*		*2.0*		*40.0*
102	3	3	3	2	1	1	0		0.0		18.5
103	**4**	**3**	**4**	**3**	**1**	**2**	**0**	×	**0.0**	=	**24.5**
104	*4*	*2*	*3*	*4*	*2*	*2*	*1*		*4.0*		*40.0*
105	*2*	*1*	*1*	*2*	*2*	*1*	*1*		*4.5*		*26.0*
106	2	1	2	2	1	2	0		0.0		16.5
107	*1*	*0*	*0*	*1*	*1*	*0*	*1*		*5.0*		*15.5*

如果需要，可以花一点儿时间看看矩阵乘是如何进行的（http://en.wikipedia.org/wiki/Matrix_multiplication）。共现矩阵和一个用户向量的乘积结果是一个向量，它的维度等于项目的个数。可以从结果向量R中的值中直接得到推荐结果：在R中最大的值对应于最佳的推荐。

表6-2显示了这个小样本数据集为用户3所做的乘积，以及结果矩阵R。可以忽略R中物品101、104、105和107对应行的值（表中为斜体），因为它们不适用于推荐：用户3已经表达过对这些物品的偏好。在余下的物品中，物品103的条目具有最高值24.5，因此是最佳推荐，其次是102和106。

6.2.4 解读结果

让我们停下来思考一下在上一节发生了什么。为什么在R中较高的值对应于更好的推荐？计算R中的每个条目近似于为每个物品估计一个偏好，但是它们为什么会近似于估计的偏好值呢？

回顾一下这个计算，R中的第3个条目为矩阵中第3行的向量与列向量U3的点积。这是两个向量中每组对应条目对之间的乘积之和：

```
4(2.0) + 3(0.0) + 4(0.0) + 3(4.0) + 1(4.5) + 2(0.0) + 0(5.0) = 24.5
```

第三行包含了物品103和所有其他物品之间的共现关系。直观而言，如果物品103和那些用户3表达过偏好的物品存在共现关系，那么它就有可能是用户3所喜欢的物品。前面的公式将共现关系和偏好值的乘积相加。当物品103总是与用户很喜欢的物品同时出现，这个相加结果就包含了大的共现值和大的偏好值之间的乘积。这会使得总和（R中条目的值）更大。这就是R中较大的值会对应于好推荐的原因。

注意，R中的值并不代表一个估计偏好值（estimated preference value）——它们相对于1而言实在太大了。理想情况下，应该利用一些额外的信息将它们归一化为估计偏好值。但是从我们所要达成的目标来看，归一化没有必要，因为重要的是推荐的顺序，而不是排序所依赖的确切值。

6.2.5 分布式实现

这个算法的确很好，但它是否也更适合于大规模的分布式实现呢？

这个算法各个组件每次仅处理全部数据的一个子集。例如，生成用户向量只是为一个用户搜集全部的偏好值并构建出一个向量。统计共现关系只需要每次检查一个向量。计算作为结果的推荐向量仅需每次加载矩阵的一行或一列。而且，许多计算只是把相关的数据高效地搜集到一起，例如从各自的偏好值中创建用户向量。

MapReduce范式正是为拥有这些特征的计算而设计的。

6.3 基于 MapReduce 实现分布式算法

现在，算法可被转换为基于MapReduce与Apache Hadoop实现的形式。如前所述，Hadoop是一个流行的分布式计算框架，主要包含两个组件：HDFS（Hadoop Distributed Filesystem，Hadoop分布式文件系统）和一个MapReduce范式的实现。

本节稍后将逐一介绍MapReduce的几个阶段，它们共同构成一条生成推荐结果的流水线。每个阶段完成一部分工作。我们将看到每个阶段的输入、输出和用途。阅读本书后续内容可知，即便是这种简单的推荐算法也需要5个MapReduce阶段，而在Mahout中这还只是最简单的形式。本节最后会给出一个完整的基于Hadoop的端到端推荐系统。



注意本章会使用Hadoop框架0.20.2版本中的API。本节代码的完整版可以在Mahout中找到，它们可在Hadoop 0.20.2或0.20.x分支中较新的版本中运行。特别需参考`org.apache.mahout.cf.taste.hadoop.item.RecommenderJob`，它调用了所有后续过程的实现。

6.3.1 MapReduce简介

MapReduce是一种思考和组织计算的方法，据此可将计算合理分布到许多机器上。MapReduce计算的形式如下：

(1) 输入的形式为许多键值对（K1,V1），通常是一个HDFS实例的输入文件；

(2) Map函数作用于每个（K1,V1）对，得到0个或多个与之不同的键值对（K2，V2）；

(3) 为每个K2合并所有的V2；

(4) 为每个K2及其对应的V2调用Reduce函数，得到0个或多个另一种不同的键值对（K3，V3），输出返回到HDFS。

这似乎是一个奇怪的计算模式，但是许多问题的处理可以套用这个模式，或由多个基于该模式的计算串在一起来完成。以这种形态为架构的问题就可以借助于Hadoop和HDFS有效地进行分布。

如果不熟悉Hadoop，可以阅读并运行Hadoop的简短教程，0.20.2版本的文档（http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html）会教给你在Hadoop中运行MapReduce作业的基本操作。

6.3.2 向MapReduce转换：生成用户向量

在这个案例中，计算将含有链接的数据文件作为输入。它的行不采用`userID,itemID,preference`的形式，而是采用`userID: itemID1 itemID2 itemID3...`的形式。这个文件放在HDFS上以便供Hadoop使用——更多实现细节会在几节之后讨论。

第一个MapReduce会形成用户向量。

- 输入文件被框架视为(`Long,String`)对，这里`Long`型的键是文件中的位置，而`String`型的值为文件中的文本行。例如`239 / 98955: 590 22 9059`。
- 每一行被map函数解析为一个用户ID和几个物品ID。该函数输出新的键值对：用户ID及其对应的物品ID，这样每个物品ID都有一个用户ID。例如`98955 / 590`。
- 框架为每个用户ID将所有对应的物品ID搜集到一起。
- Reduce函数利用全部的物品ID为该用户构造一个向量（`Vector`），并输出这个用户的ID，与该用户的偏好向量相对应。该向量中的值均为0或1。例如`98955 / [590:1.0, 22:1.0, 9059:1.0]`。

该想法的一个实现可见代码清单6-1和代码清单6-2，其中实现了Hadoop MapReduce中的`Mapper`和`Reducer`接口。这是典型的MapReduce计算过程，包含这样一对相关类的实现。这就是实现前述流程所需的全部内容，剩下的部分由Hadoop负责。

代码清单6-1 解析Wikipedia链接文件的Mapper

```
public class WikipediaToItemPrefsMapper
    extends Mapper<LongWritable,Text,VarLongWritable,VarLongWritable> {

  private static final Pattern NUMBERS = Pattern.compile("(\\d+)");

  public void map(LongWritable key,
                  Text value,
                  Context context)
      throws IOException, InterruptedException {
    String line = value.toString();
    Matcher m = NUMBERS.matcher(line);
    m.find();                                            定位用户ID
    VarLongWritable userID =
        new VarLongWritable(Long.parseLong(m.group()));
    VarLongWritable itemID = new VarLongWritable();
    while (m.find()) {
      itemID.set(Long.parseLong(m.group()));
      context.write(userID, itemID);                     为每个物品ID生成用户-物品对
    }
  }
}
```


代码清单6-2 从用户的物品偏好中生成Vector的Reducer

```
public class WikipediaToUserVectorReducer extends
  Reducer<VarLongWritable,VarLongWritable,VarLongWritable,VectorWritable> {
  public void reduce(VarLongWritable userID,
                     Iterable<VarLongWritable> itemPrefs,
                     Context context)
      throws IOException, InterruptedException {
    Vector userVector = new RandomAccessSparseVector(
        Integer.MAX_VALUE, 100);
    for (VarLongWritable itemPref : itemPrefs) {          循环遍历用户的物品-偏好对
      userVector.set((int)itemPref.get(), 1.0f);          在“物品ID”维设置物品偏好值
    }
    context.write(userID, new VectorWritable(userVector));
  }
}
```

为了满足说明的需要，这里仅给出Mahout实际实现的简化版。它们不包含优化和配置选项，但是均可运行并输出有用的结果。

6.3.3 向MapReduce转换：计算共现关系

下一步计算过程为另一个MapReduce，它使用第一个MapReduce的输出来计算共现关系。

(1) 输入是用户ID及对应的用户偏好`Vector`（上一个MapReduce的输出），例如`98955 / [590:1.0,22:1.0,9059:1.0]`。

(2) Map函数根据用户的偏好来决定所有的共现关系，并为每一个共现关系生成一个物品ID对——物品ID对应到物品ID。无论是从一个物品ID到另一个的对应关系，或截然相反的对应关系，

都会被记录下来。例如590/22。

(3) 框架为每个物品搜集与之对应的所有共现关系。

(4) Reducer为每个物品ID统计它收到的全部共现关系，并构造一个新的Vector，通过统计它们共现的次数来表达一个物品的全部共现关系。它们可以当做共现矩阵的行或列使用。例如590/[22:3.0,95:1.0,...,9059:1.0,...]。

这个阶段的输出实际上是共现矩阵。代码清单6-3和代码清单6-4给出了一个在Hadoop的Mahout中的简单实现。这里同样有Mapper和Reducer的相应实现。

代码清单6-3　计算共现关系的Mapper

```
public class UserVectorToCooccurrenceMapper extends
    Mapper<VarLongWritable,VectorWritable,IntWritable,IntWritable> {
  public void map(VarLongWritable userID,
                  VectorWritable userVector,
                  Context context)
      throws IOException, InterruptedException {
    Iterator<Vector.Element> it =
      userVector.get().iterateNonZero();                    仅循环遍历非零元素
    while (it.hasNext()) {
      int index1 = it.next().index();
      Iterator<Vector.Element> it2 =
        userVector.get().iterateNonZero();
      while (it2.hasNext()) {
        int index2 = it2.next().index();
        context.write(new IntWritable(index1),
                      new IntWritable(index2));           记录项目ID
      }
    } }
```

代码清单6-4　计算共生关系的Reducer

```
public class UserVectorToCooccurrenceReducer extends
    Reducer<IntWritable,IntWritable,IntWritable,VectorWritable> {
  public void reduce(IntWritable itemIndex1,
                  Iterable<IntWritable> itemIndex2s,
                  Context context)
      throws IOException, InterruptedException {
    Vector cooccurrenceRow =
      new RandomAccessSparseVector(Integer.MAX_VALUE, 100);
    for (IntWritable intWritable : itemIndex2s) {
      int itemIndex2 = intWritable.get();
      cooccurrenceRow.set(
        itemIndex2,
        cooccurrenceRow.get(itemIndex2) + 1.0);           累加物品1和2的共现次数
    }
    context.write(
        itemIndex1,
        new VectorWritable(cooccurrenceRow));             记录完整的物品1共现向量
  }
}
```

6.3.4 向MapReduce转换：重新思考矩阵乘

现在可以使用MapReduce将第一步中得到的用户向量和第二步中的共现矩阵相乘，从而得到一个推荐向量，从中算法可以推测出推荐结果。

但是在这里，这个乘法可以用一种不同但更为高效的方法来做——一种更适合MapReduce计算的形式。这个算法不会使用传统的矩阵乘，那是让每一行都去乘用户向量（作为一个列向量），以生成结果R中的一个元素。

```
for 共现矩阵中的每一行i
    计算行向量i和用户向量的点积
    将点积结果存入R中第i个元素
```

这个算法我们在学校都学过，为什么不用它呢？问题都出在性能上，这里我们正好用它来理解一下设计大型矩阵和向量操作时应如何思考，以便获得适当的性能。传统的算法会用整个共现矩阵，因为它需要对每一行做一次向量的点积。这里任何对全部输入进行处理的算法都很"糟糕"，因为输入可能会超级庞大，甚至无法本地化。与此相反，矩阵乘可以转化为一个对共现矩阵中列的函数。

```
将R置为空向量
for 共现矩阵中的每个列i
    将列向量i和用户向量中的第i个元素相乘
    将这个向量加到R上
```

请花点儿时间想想这个方法，可以用一个小例子来试一下，这也是一个正确的矩阵乘。此时仍算不上是改进，因为它还是要按列对整个共现矩阵进行处理。

但是，只要用户向量中的元素i为0，循环就可以完全被跳过去，因为乘积是零向量并且不会影响结果。于是只要对用户向量的非零元素执行循环即可。列数等于用户给出偏好的个数，当用户向量稀疏时，它远小于列的总数。

按此方法，算法可以有效地对计算进行分布。可以将列向量i输出到所有与之相乘的元素上。乘积可以彼此独立地进行计算和存储。

6.3.5 向MapReduce转换：通过部分乘积计算矩阵乘

从前面的步骤中可以获得共现矩阵的列。因为这个矩阵是对称的，行与列相同，所以输出在理论上可以被看做行，也可以被看做列。这些列将物品ID作为键，算法必须将所有用户向量中的每一列去和该物品中的每一个非零的偏好值相乘。就是说，它必须将物品ID逐一和用户ID以及偏好值对应起来，并在Reducer中将它们汇聚在一起。在将每个值都与这个共现矩阵的列相乘之后，就会生成一个向量，形成面向用户的推荐向量R的一部分。

这里的难点在于在一个计算过程中要合并两种不同的数据：共现列向量和用户偏好值。这原本在Hadoop上是不可能实现的，因为在Reducer中的值只能为Writable这一种类型。有一种巧妙的实现可以解决这个问题，即构建一个新的Writable，即VectorOrPrefWritable，它含有一种或另一种数据类型。虽然可能有点儿取巧，但在设计分布式计算时，为支持优雅而高效的计

算而修改一些规则是非常有用的，也是必须的。

这里的Map阶段实际包含了两个Mapper，每个产生不同类型的Reducer输入。

- 第一个Mapper的输入为共现矩阵：以物品ID为键，对应于Vector形式的列。例如590 / [22:3.0,95:1.0,...,9059:1.0,...]。

  Map函数简单地转发其输入，但形式上采用以VectorOrPrefWritable封装的Vector。
- 第二个Mapper的输入为用户向量：以用户ID为键，对应于Vector形式的偏好值。例如98955/[590:1.0,22:1.0,9059:1.0]。

  对于用户向量中的每一个非零值，Map函数输出一个物品ID，及对应的用户ID和偏好值，以VectorOrPrefWritable的形式封装。例如590 / [98955:1.0]。

  框架按照物品ID将共生关系列和所有的用户ID-偏好值对汇聚在一起。

  reducer将这些信息归并为一条输出记录并存储下来。

代码清单6-5显示了以VectorOrPrefWritable形式封装的共现关系列。

代码清单6-5　封装共现关系列

```
public class CooccurrenceColumnWrapperMapper extends
    Mapper<IntWritable,VectorWritable,
           IntWritable,VectorOrPrefWritable> {
  public void map(IntWritable key,
                  VectorWritable value,
                  Context context) throws IOException, InterruptedException {
    context.write(key, new VectorOrPrefWritable(value.get()));
  }
}
```

在代码清单6-6中，用户向量被分割为其独立的偏好值和输出（根据物品ID，而非用户ID）。

代码清单6-6　分割用户向量

```
public class UserVectorSplitterMapper extends
    Mapper<VarLongWritable,VectorWritable,
           IntWritable,VectorOrPrefWritable> {
  public void map(VarLongWritable key,
                  VectorWritable value,
                  Context context) throws IOException, InterruptedException {
    long userID = key.get();
    Vector userVector = value.get();
    Iterator<Vector.Element> it = userVector.iterateNonZero();
    IntWritable itemIndexWritable = new IntWritable();
    while (it.hasNext()) {
      Vector.Element e = it.next();
      int itemIndex = e.index();
      float preferenceValue = (float) e.get();
      itemIndexWritable.set(itemIndex);
      context.write(itemIndexWritable,
                    new VectorOrPrefWritable(userID, preferenceValue));
    }
  }
}
```

从技术上讲，在这两个Mapper之后并没有真正的Reducer；我们不能把两个Mapper的输出导入一个Reducer中。相反，它们独立运行，并将输出结果传递到一个空的Reducer，最终保存在两个位置。这两个位置可以作为另一个MapReduce的输入，它的Mapper什么也不做，而Reducer将物品的一个共现关系列向量，并和该物品对应的所有用户及偏好值汇聚在一起形成一个实体，称为VectorAndPrefsWritable。上述过程在ToVectorAndPrefReducer中实现，简单起见不再赘述。

有了共现矩阵的列和用户偏好，且它们均以物品ID为键，算法就可以将它们导入到一个mapper中，并输出该列和用户偏好的乘积。这个步骤见代码清单6-7。

(1) mapper的输入是按物品组织的所有共现矩阵列和用户偏好。例如590 / [22:3.0,95:1. 0,...,9059:1.0,...]和590 / [98955:1.0]。

(2) mapper的输出是共现关系列乘以每个对应用户的偏好值。例如590 / [22:3.0,95:1.0,...,9059:1.0,...]。

(3) 框架按用户将这些乘积汇集在一起。

(4) reducer将输入的所有向量拆开后求和，形成对该用户的最终推荐向量R。例如590 / [22:4.0,45:3.0,95:11.0,...,9059:1.0,...]。

代码清单6-7　计算部分推荐向量

```
public class PartialMultiplyMapper extends
    Mapper<IntWritable,VectorAndPrefsWritable,
           VarLongWritable,VectorWritable> {
  public void map(IntWritable key,
                  VectorAndPrefsWritable vectorAndPrefsWritable,
                  Context context) throws IOException, InterruptedException {

    Vector cooccurrenceColumn = vectorAndPrefsWritable.getVector();
    List<Long> userIDs = vectorAndPrefsWritable.getUserIDs();
    List<Float> prefValues = vectorAndPrefsWritable.getValues();

    for (int i = 0; i < userIDs.size(); i++) {
      long userID = userIDs.get(i);
      float prefValue = prefValues.get(i);
      Vector partialProduct = cooccurrenceColumn.times(prefValue);
      context.write(new VarLongWritable(userID),
                    new VectorWritable(partialProduct));
    }
  }
}
```

这个mappper会写很多数据。对于每个用户-物品关联，它都会输出共现矩阵中一个完整列的副本。这几乎是必须的，这些副本要在reducer中和其他列的副本组合与相加，以生成一个推荐向量。

但是这个阶段可以引入一个优化：combiner。它就像一个小型的reducer操作（实际上，combiner也扩展了Reducer），它当map的输出仍在内存时执行，在输出记录未被执行写操作之前将几个记录合并为一个。这会节省I/O，而这种事并不常见。在这里，对一个用户输出两个向量A和B，就和对这个用户输出一个向量A+B一样——它们在最后都会被加在一起。

代码清单6-8给出了一个combiner，处理`PartialMultiplyMapper`的输出。它能节省多少I/O取决于Hadoop在写入磁盘前在内存中能够存放多少输出结果（map溢出缓冲区），以及在map节点的输出中涉及一个用户的列出现得有多频繁。也就是说，如果大量`Mapper`的输出被存放在内存中，它们很多都可以被合并，那么combiner就会节省大量的I/O。Mahout对这些作业的实现会尝试将`io.sort.mb`增大到1 GB，来为mapper的输出预留比平时更多的内存。

代码清单6-8　实现部分乘积的combiner

```
public class AggregateCombiner extends
    Reducer<VarLongWritable,VectorWritable,
            VarLongWritable,VectorWritable> {
  public void reduce(VarLongWritable key,
                     Iterable<VectorWritable> values,
                     Context context)
      throws IOException, InterruptedException {
    Vector partial = null;
    for (VectorWritable vectorWritable : values) {
      partial = partial == null ?
        vectorWritable.get() : partial.plus(vectorWritable.get());
    }
    context.write(key, new VectorWritable(partial));
  }
}
```

6.3.6　向MapReduce转换：形成推荐

最后，算法为每个用户合并推荐向量，以形成推荐结果，如代码清单6-9所示。

代码清单6-9　处理来自向量的推荐结果

```
public class AggregateAndRecommendReducer extends
    Reducer<VarLongWritable,VectorWritable,
            VarLongWritable,RecommendedItemsWritable> {

  ...

  public void reduce(VarLongWritable key,
                     Iterable<VectorWritable> values,
                     Context context)
      throws IOException, InterruptedException {

    Vector recommendationVector = null;
    for (VectorWritable vectorWritable : values) {
      recommendationVector = recommendationVector == null ?
        vectorWritable.get() :
                                                        求和以形成推荐向量
     recommendationVector.plus(vectorWritable.get());
    }

    Queue<RecommendedItem> topItems = new PriorityQueue<RecommendedItem>(
          recommendationsPerUser + 1,
          Collections.reverseOrder(
```

```
            ByValueRecommendedItemComparator.getInstance()));

    Iterator<Vector.Element> recommendationVectorIterator =
        recommendationVector.iterateNonZero();
    while (recommendationVectorIterator.hasNext()) {
      Vector.Element element = recommendationVectorIterator.next();
      int index = element.index();
      float value = (float) element.get();
      if (topItems.size() < recommendationsPerUser) {
        topItems.add(new GenericRecommendedItem(
                               indexItemIDMap.get(index), value));
      } else if (value > topItems.peek().getValue()) {
        topItems.add(new GenericRecommendedItem(
                               indexItemIDMap.get(index), value));
        topItems.poll();
      }
    }

    List<RecommendedItem> recommendations =
      new ArrayList<RecommendedItem>(topItems.size());
    recommendations.addAll(topItems);
    Collections.sort(recommendations,
                     ByValueRecommendedItemComparator.getInstance());
    context.write(
        key,
        new
     RecommendedItemsWritable(recommendations));
  }
}
```

找到最大的*N*个值

按序输出推荐结果

输出最后以一个或多个压缩文本文件的形式存放在HDFS上。该文本文件的行采用如下形式：

```
3    [103:24.5,102:18.5,106:16.5]
```

每个用户ID之后跟随一个以逗号分隔的物品ID列表（物品ID后跟着一个冒号和推荐向量中的对应条目，不论其是否有用）。应用可以从HDFS中获取、解析与使用这个输出结果。注意从Mahout中获得的输出结果都是gzip压缩过的，这样做是为了节省空间。

最终，我们用来自Wikipedia数据集的原始输入数据获得了推荐结果。如你所见，即便是这样简单的推荐程序，在Hadoop这样的分布式系统上做设计和实现时，都会如此不同。即便是简化的形式，它仍包含5个阶段，每个阶段执行其中一部分计算或转换。下一步自然是将它们全都运转起来，来更好地理解推荐是如何实际工作的。

6.4 在 Hadoop 上运行 MapReduce

现在是时候在Wikipedia链接数据集上运行这个实现了。虽然Hadoop有能力在上千台机器的集群上运行，但是你首先会在只有一台机器的集群上运行Hadoop计算：你的本机。相比于一个适当的集群，学习在本地计算机上建立Hadoop集群要简单得多。

这个过程不会和使用一个真实集群有很大差别。首先，你需要在本机上安装一个伪分布式的Hadoop集群，然后就可以在上面使用Mahout运行MapReduce作业。

6.4.1　安装Hadoop

如第1章所述，你需要从Apache下载一个最新的Hadoop副本（http://hadoop.apache.org/common/releases.html）。本书写作时，我们推荐使用的版本为0.20.2。遵照http://hadoop.apache.org/common/docs/current/single_node_setup.html的安装指导，配置为伪分布方式。

在使用bin/start-all.sh运行Hadoop的守护进程之前，要在conf/mapred-site.xml中做一个修改，增加一个名为mapred.child.java.opts的属性，值为-Xmx1024m。这会让Hadoop的worker能够使用1 GB的堆空间。一旦你开始运行所有的Hadoop守护进程，就可以结束安装。

你现在可以在本机上运行一个包含HDFS实例的完整Hadoop集群。输入要放在HDFS上，这样才能为Hadoop所用。你也许会感到奇怪，既然数据已经在本地文件系统上了，为什么还要再复制到HDFS上。回顾一下，通常Hadoop是一个运行在许多机器上的框架，因此它使用的任何数据都应该能被多台机器看见，而不是一台。HDFS能够让多台机器获得数据。使用如下命令复制输入到HDFS上：

```
bin/hadoop fs -put links-simple-sorted.txt input/input.txt
```

要为数据集中的每篇文章都生成推荐会花费很长的时间，因为只有一台机器运行，还引入了分布式计算框架的开销。但你可以决定Mahout的推荐次数，比如说仅为一个用户（文章）进行计算。创建一个只有一行的文件，包含数字3，并将其存为users.txt。这就是算法会为之生成推荐的文章列表，在测试中只有一篇文章。把它放入HDFS实例中，命令如下：

```
bin/hadoop fs -put users.txt input/users.txt
```

6.4.2　在Hadoop上执行推荐

org.apache.mahout.cf.taste.hadoop.item.RecommenderJob 将各种 Mapper 和 Reducer组件粘连在一起。你可以在Mahout源发布中找到它。它配置并调用我们之前讨论的一系列MapReduce作业。这些MapReduce及其之间的关系如图6-2所示。

为了运行RecommenderJob并让Hadoop来运行这些作业，你需要将所有这些代码及其依赖的所有代码打包为一个JAR文件。这很容易通过在Mahout发布包的core/目录下运行mvn clean package来完成——生成一个类似mahout-core-0.5-job.jar这样的文件。另外，你也可以使用Mahout发布包中预编译的作业JAR包。

现在，开始运行这些命令：

```
hadoop jar mahout-core-0.5-job.jar \
  org.apache.mahout.cf.taste.hadoop.item.RecommenderJob \
  -Dmapred.input.dir=input/input.txt \
  -Dmapred.output.dir=output --usersFile input/users.txt --booleanData
```

Hadoop会接手并开始运行这一系列的作业。这会花上几个小时，因为只有一台机器（你的本机）在处理这个计算。即使通过一组机器，也不能期待能在几分钟内完成任务；初始化集群的开销、分布数据并执行代码，还有整理结果都会花很多时间。

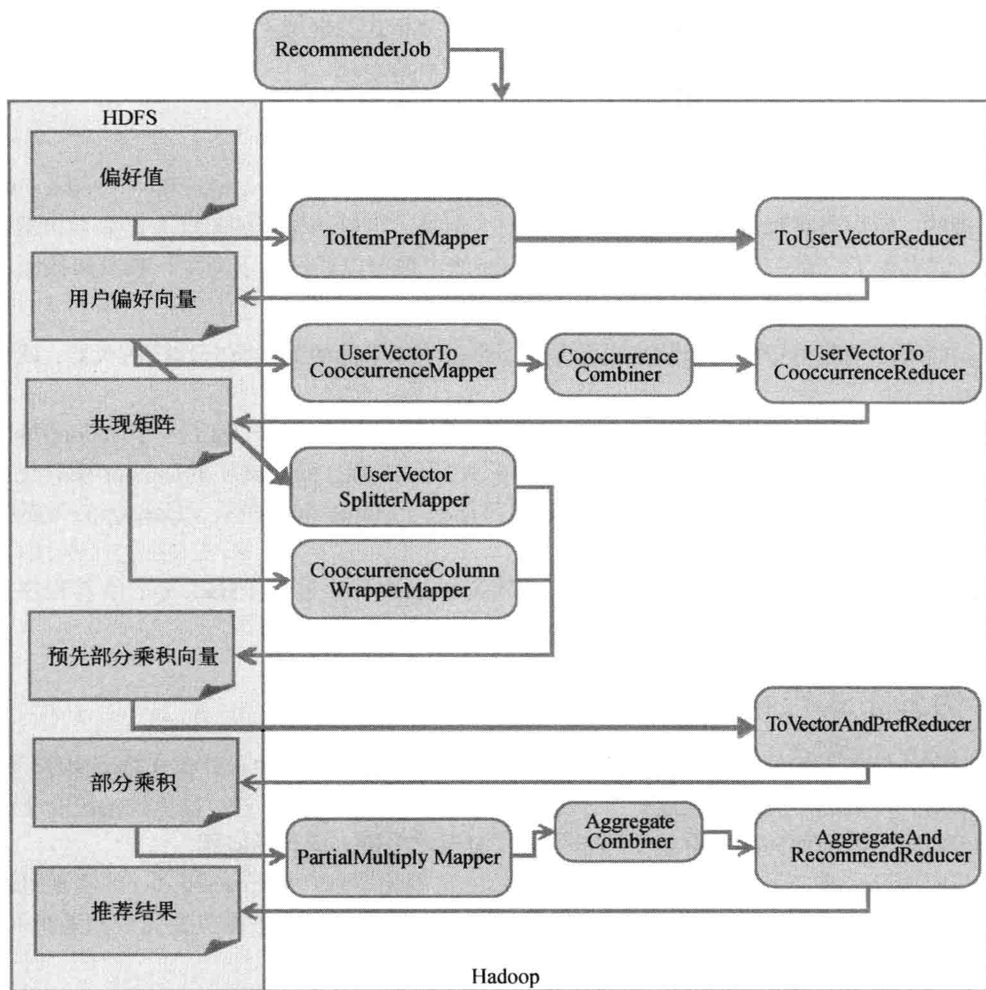


图6-2　RecommenderJob之间的关系、它调用的MapReduce，以及向HDFS读写的数据

实际上，若在单机集群上为这个数据集的全部570万用户进行推荐，这会花费大约700小时的CPU时间，每次推荐大概用半分钟。运行时间取决于最后的阶段，那里会有向量乘和向量加的操作。时间开销会比在规模较小的非分布式推荐程序上还多，只是还到不了几倍那么高。依据当前在Amazon Elastic MapReduce上运行虚拟实例的价格，每1000个用户会花费大约0.01美元。当然，不必花费700小时来计算所有这些推荐；如果部署了100个工作节点，整个过程用7个多小时就能完成。

回到本机的实例，如果你耐心地等待它完成对一个用户的推荐，就会在HDFS的output/目录下找到结果。结果存在一个名为part-r-00000的文件中。

使用命令`bin/hadoop fs -get output/part-r-00000`将结果复制到本地文件系统，就可以访问并使用这个文件了。

庆祝一下，这样就完成了！你已经利用一个完全分布式的框架（单节点集群）生成了推荐结果。最后，不要忘记通过bin/stop-all.sh来关闭Hadoop。

6.4.3 配置mapper和reducer

这里有一个要点。以前，我们让Hadoop默认每次仅运行一个map worker和一个reduce worker。这是合理的，因为当时算法仅在一台单机上运行。但是，当启动作业的集群有许多台机器时，一个worker就太少了。在一个真实的集群中，worker的个数可以通过如下命令行参数来控制：

```
-Dmapred.map.tasks=X -Dmapred.reduce.tasks=Y
```

开始时可以用集群中核的个数来设置X和Y。例如，如果你的集群有5台4核的机器，就将它们均设为20。

至此，你已经成功地在Hadoop上设计实现了一个分布式算法、安装了一个Hadoop集群，并让实现在Hadoop上运行起来。这就是所需学习的大部分内容，而与真实的Hadoop集群交互本质上与此相同。这还会让你能够使用Mahout处理其他基于MapReduce和基于Hadoop的机器学习算法，它们会在以后的章节中出现。

在结束对推荐程序的讨论之前，我们再来看Mahout中另一种基于Hadoop的推荐程序，它是一种介于分布式和非分布式方法之间的混合物。

6.5 伪分布式推荐程序

我们之前描述了如何在单机上通过Mahout创建、测试并操作各种非分布式推荐引擎。本章说明如何采用一种截然不同的方式运行一个完全分布式的推荐程序。不过，这里的情况介于分布式和非分布之间，因为面向的是希望使用非分布式实现在多机上运行的应用。

这些应用已经在非分布式框架下开发出高效的定制化实现。Recommender的实现也许与所有的非分布式实现一样，是与DataModel直接绑定的，需要对所有数据进行高效和随机的访问。因此，我们很难或无法按照完全分布的方式来改造它。

Mahout为此提供了一个伪分布式的推荐引擎框架。这只是一个对Hadoop的应用，Hadoop可以并行地运行给定推荐引擎的多个独立、非分布式的实例。因此，我们可以轻松地让一个成熟的、非分布式算法使用许多机器。这种机制实际上并没有将计算并行化，它只是管理了多个非分布式实例的操作。性能与直接运行一个非分布式的实例相同，但是它允许你在n台机器上运行n个推荐程序，每个处理$1/n$的推荐，所用时间也就变成单机的$1/n$。

这种方法的缺点是可扩展性受限：一个非分布式的计算无法处理更大量的数据，这是由运行它的机器的资源所决定的——一个不适合在一台大机器上做的运算转到n个独立的大机器上运行也不会适合。伪分布不会改变这一点。

这里没有引入新的算法和代码；Mahout的伪分布式推荐引擎框架仍用原来的Recommender，但是在Hadoop之上运行。从概念上看，它使用Hadoop将用户集合分在n台机器上，并把输入数据复制到每台机器，然后在每台机器上运行Recommender对一个用户子集进行推荐。

这个过程和以前没有什么两样。安装和运行Hadoop，把包含偏好的输入文件复制到HDFS。如果你希望尝试这个框架，选择一个输入，比如GroupLens 100 K数据集中的ua.base。（Wikipedia链接数据集太大而无法用于非分布式实现。）这个数据集中的输入文件ua.base需要做一个小转换才能被这段代码处理，即制表符必须转换为逗号。你可以用自己喜欢的编辑器来做这件事，或者使用如下Unix命令：

```
cut -f1-3 ua.base | tr '\t' ',' > ua.base.hadoop
```

将ua.base.hadoop放入HDFS，例如input/ua.base.hadoop。

框架需要知道Recommender实现的名称，以便对它初始化和使用。对于这个实现只有一个要求，即它必须提供一个包含唯一参数DataModel的构造函数。有了这个构造函数，框架就可以做余下的工作。通常情况下，你会在这里提供一个为自己的应用定制的Recommender。但在测试时，用SlopeOneRecommender就可以，因为它就可以仅通过DataModel参数进行初始化。

文件mahout-core-0.5-job.jar已经包含了SlopeOneRecommender，因为它是一个标准的Mahout实现。但是如果使用的是自己的实现，你就需要把它及其依赖的类放在JAR文件里。最后用如下命令来完成：

```
jar uf mahout-core-0.5-job.jar -C [classes directory]
```

其中，类目录（class directory）存放由IDE或其他构建工具编译出的代码。

最后，运行这个作业：

```
hadoop jar mahout-core-0.5-job.jar \
  org.apache.mahout.cf.taste.hadoop.pseudo.RecommenderJob \
  -Dmapred.input.dir=input/ua.base.hadoop \
  -Dmapred.output.dir=output \
  --recommenderClassName \
  org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender
```

你同样会在HDFS的output/目录下看到输出结果。

这就是全部过程；如果输入数据未达到需要真正分布式算法的规模，伪分布式推荐框架是一种又快又好的办法，可以利用更多的计算能力来更快地获得推荐结果。

6.6 深入理解推荐

本章所介绍的Mahout推荐引擎的分布式部分仍在开发中，所以阅读时请参考最新的文档和代码，关注Mahout以获悉它的变化。Mahout算是一个“半成品”，由一个不断成长的社区开发与维护；我们的目标是不断地完善或增强对推荐程序和机器学习的领悟和有效使用能力。

在探讨聚类和分类算法之前，我们将简要而明晰的想法归结起来，作为后续思考和了解推荐引擎的起点。

6.6.1 在云上运行程序

你找不到100台机器运行这些大型分布式计算？没关系，如今服务提供商允许你从一个计算

云上租用存储和计算时间。

亚马逊的Elastic MapReduce服务（http://aws.amazon.com/elasticmapreduce/）就是这样一个服务，如图6-3所示。它使用亚马逊的S3存储服务（而不是一个单纯的HDFS实例）在云上存储数据。上传JAR文件和数据到S3之后，你就可以使用它们的AWS Console调用一个分布式计算，只需提供过去在命令行下调用计算时使用的相同参数。

DEFINE JOB FLOW　SPECIFY PARAMETERS　CONFIGURE EC2 INSTANCES　BOOTSTRAP ACTIONS

Please review the details of your job flow and click "Create Job Flow" when you are ready to l

Job Flow Name: My Job Flow
Type: Custom Jar

Jar Location: s3://my-bucket/mahout.jar
Jar Arguments: org.apache.mahout.cf.taste.hadoop.item.RecommenderJob --Dmapred.map.tasks=10 -Dmapred.reduce.tasks=10 --

Number of Instances: 10
Type of Instance: m1.small
Amazon EC2 Key Pair:
Amazon S3 Log Path:
Enable Hadoop Debugging: No

Bootstrap Actions: No Bootstrap Actions created for this Job Flow

‹ Back　Create Job Flow

图6-3　亚马逊的AWS Elastic MapReduce控制台

进入AWS Console主界面之后，选择Amazon Elastic MapReduce栏，再选择Create New Job Flow（创建新作业流）选项。给这个新的流（flow）命名，并指定Run Your Own Application（运行自己的应用程序）。选择Custom Jar（自定义Jar）类型并继续。指定S3上JAR文件的放置位置；形式为S3:URL，类似于s3://my-bucket/target/mahout-core-0.5-job.jar。

作业的参数和在命令行下运行时一样；当然在这里配置mapper和reducer的个数是非常必要的。mapper和reducer的个数可以依你的喜好来定；和以前一样，开始时可以设置为你为计算预留的虚拟核的个数。虽然可以使用任意的实例类型，但除非有特别的原因，否则开始时都用一个常规类型：small、large或extra-large。实例的个数和类型在下一个AWS Console界面中选择。

如果输入数据非常大，一些推荐作业（如在`org.apache.mahout.cf.taste.hadoop.item`中的作业）可能需要为每个mapper或reducer分配更多的RAM。此时，你也许不得不选择一个high-memory实例类型。你可能还需要用到high-CPU实例类型；风险是作业会花费过多的时间从S3中读写数据来喂饱这些贪婪的CPU，否则它们大部分时间会“很饿”。因此，传统的实例类型

是一个很好的起点。如果你正在使用small实例类型，其中每个实例拥有一个虚拟核，就可以设置mapper和reducer的个数为你选择的实例个数。

你可以保持其他参数不动，除非有理由变更它们。

这里介绍了在Elastic MapReduce上运行一个推荐作业的基础内容；参考亚马逊的文档可以了解更多关于监视、停止和调试这些作业的信息。

6.6.2 考虑推荐的非传统用法

我们对推荐程序的讨论即将收尾。但在探讨聚类之前，有必要转换一下思路，最后讨论一下推荐程序在你项目中的适用性。

虽然Mahout推荐引擎的API是用“用户”和“物品”表述的，但这个框架并不假设“用户”就是人，也不假设“物品”就是书或DVD这样的物品。例如，推荐引擎可应用在约会网站的数据上为某个或某些人推荐某个或某些人。还可以怎样使用推荐引擎呢？下面这些思路可以引发你的思考。

- **为物品推荐用户** 通过物品ID和用户ID的交换，推荐引擎的输出转变为：哪些用户会对指定物品更感兴趣。
- **扩展物品的范畴** 给定与用户关联的地方、时间、使用模式或者其他人，就可以为之推荐相同类型的物品（地方、时间、使用模式或者其他人）。
- **找到最相似的物品** Mahout中基于物品的推荐程序实现使得发现一组最相似的物品变得很容易。
- **扩展偏好值的范畴** 通常无法从用户那里获得明确的偏好值。你只能根据所了解的用户对事物的关系来推测。
- **考虑不止一个用户和物品** 可以为一对用户进行推荐，即把一对用户视为一个用户。你还可以把物品及其位置统一视为一个物品来进行推荐。

Mahout没有为这些应用案例提供专门的支持，但它们都可以在Mahout之上实现。这也许是Mahout未来的一个发展方向，也或许会出现在一些专用的第三方项目中。需要特别指出的是，基于用户的行为和其他数据来推测隐含的评分，这本身就很有趣且很重要，但非Mahout所关注的内容。

6.7 小结

本章我们简要审视了来自于维基百科文章链接的大型数据集。它庞大的1.3亿偏好值需要一个不同的分布式推荐生成方法。

我们权衡了从单机非分布式算法向集群分布式计算过渡的过程。接着我们简要介绍了MapReduce范式及其在Hadoop上的实现，它是管理这种分布式计算的一种手段。

我们将之前基于物品的推荐算法转换为一种不同的分布式实现，后者依赖于矩阵和向量操作来发现最佳推荐。我们再次使用Wikipedia数据集，让它可在Hadoop中使用，并在本地Hadoop和

HDFS实例上为该数据集生成推荐。

最后，我们用Mahout做了伪分布式推荐的计算：在Hadoop上运行了非分布式`Recommender`实现的几个独立实例。

这就是Mahout上的推荐引擎。至此，我们有序地介绍了机器学习中一个方面，从小输入和非分布式计算逐步过渡到了大规模的分布式计算。现在，我们将转而讨论Mahout上的聚类和分类，它们涉及更多复杂的机器学习理论，并会更多用到分布式计算。有了推荐引擎，你已经为此做好了准备。请继续读下去。

Part 2

第二部分

聚　类

本书这一部分，贯穿第7章到第12章，讨论Apache Mahout中的聚类算法。利用这里描述的技术，我们可以把相近的数据片段归为一个集合或者簇。聚类有助于揭示大规模数据中信息的有趣组合。

这一部分首先讲述聚类中的简单问题，给出了一些Java示例。之后，你会看到更多在真实场景下的示例，并学会让Apache Mahout以Hadoop作业的方式运行，以便轻松地对大量数据进行聚类。

第7章介绍了聚类的定义，并通过一个对二维平面上点的聚类示例展开阐释。第8章介绍了向量的概念，并解释了如何使用它们表示数据。第9章介绍了Apache Mahout中实现的各种聚类算法。这一章通过清晰的例子展示了各种聚类算法是如何适应各种应用场景的。

第10章介绍如何评估聚类，以及如何通过调节Mahout中的各种选项和参数改进聚类质量。第11章讨论了Apache Mahout聚类的分布式实现，解释了聚类如何以Hadoop作业的形式运行以处理大数据集合。

最后，第12章基于上述各章的内容探讨了一些现实场景下的聚类问题，以及基于Apache Mahout的解决方案。

第 7 章

聚类介绍

本章内容

- 初识聚类
- 相似性概念的理解
- 在Mahout中运行简单的聚类示例
- 用于聚类的各种距离测度方法

人类倾向于跟志趣相投的人在一起，也就是"物以类聚，人以群分"。我们有足够的智力寻找重复的模式，我们不断将看到的、听到的、闻到的和品尝到的与记忆中已经存在的东西相联系。举例来说：蜂蜜的味道提示我们它尝起来像糖而不是盐，所以我们把味道像糖和蜂蜜的东西归为甜食，即使不知道甜食尝起来是什么样，我们也知道世界上所有有甜味的东西都是类似的，可以归为同一类。我们也知道它们与那些咸的东西有多大不同。我们会不自觉地把味道归为这种簇（cluster），这样就有了甜味和咸味的簇，每个簇中都包含上百种东西。

在自然界，我们可以观察到其他很多种群。例如猿和猴子，它们同属于灵长类。但所有的猴子都具有相同的特征，如较矮的身材、长尾巴和扁平的鼻子，而猿则具有更大的体型、更长的胳膊和更大的脑袋。猿和猴子有不同的外貌，但它们都喜欢香蕉。因此，我们既可以认为猿和猴子是两种的种群，也可以认为它们都属于喜欢香蕉的灵长类动物。是否可以将事物归成一个簇，这完全取决于我们在考量它们之间相似性时所选择的特征参数（在这个例子里则是灵长类）。

这一章，你将了解聚类是什么，聚类是怎样与数学中的概念相联系的，并看到Mahout中的聚类示例。现在，让我们从了解基本概念开始吧。

7.1 聚类的基本概念

那么，聚类过程是怎样的呢？假如你掌管着一个有几千本藏书图书馆的钥匙，而这些书的摆放又毫无规律可言。那么进入图书馆的读者就不得不从所有的书中一本一本地去找他想要的书。这样不仅非常麻烦低效，而且很乏味。

如果把这些书按书名字母顺序来分类，这对于通过书名来查找书籍的读者会有很大帮助。但如果大多数人都只是想简单地浏览图书，或是想研究一个很笼统的主题，这时该如何应对呢？因

此，按照主题对书籍分类会比按照书名字母排序更有用。但是这种分类该从何做起呢？假如你刚刚接手这项工作，甚至还不知道这些书都涉及什么内容——网上冲浪、言情小说，亦或讨论的是一些你从未遇到过的主题。

要实现基于主题的分类，你可以把所有书籍摆成一排，然后一本一本的读。直到读到一本书并发现它的内容与之前某本书相似，你就可以把这本书放过去，将它们堆在一起。当你把所有书全部读完以后，就会有几百堆书，而不是几千本散放的书。

非常好！这是你关于聚类的第一次体验。如果你觉得上百个主题太多了，可以回到起点，重复前面把书分成堆的过程，直至书堆的主题之间具有足够的差异。

聚类就是将一个给定文档集（collection）中的相似项目分成不同簇的过程。我们可以将这些簇看做一组簇内相似而簇间有别的项目的集合。

对文档集聚类涉及如下三件事。

- **一个算法** 即将书组织在一起的方法。
- **相似性和不相似性的概念** 在前面的讨论中，我们依赖于你对这些书的判断，即哪些书属于一个已有的书堆，或是否应该开始弄一个新的书堆。
- **停止的条件** 在这个图书馆的例子中，有一个关键节点，即在此之后，书不能再加入书堆，或者这些书堆之间已经具有明显不同的主题。

目前为止，我们认为对项目聚类就是把它们堆在一起，但实际上就是对它们进行了分组。而从概念上看，聚类更接近于寻找哪些项目可以形成相近的组，然后把它们圈起来。图7-1给出了在一个标准x–y平面上点的聚类示意图，每个圆圈代表一个簇，每个簇包含了多个点。

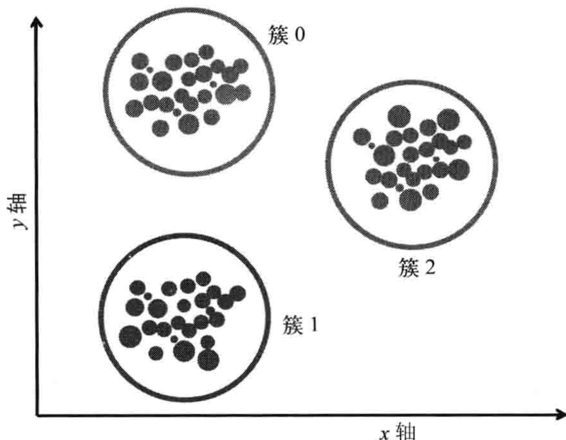


图7-1 在x–y平面上的点。圆圈代表簇，而平面上的点可视为三个逻辑组。聚类算法有助于确认这些组

在这个简单示例中，圆圈清晰地展示了一种最佳聚类，它基于距离远近，将点划分成三个簇。这些圆圈能让我们很好地理解簇，这是因为簇也是由中心点和半径来定义的。圆圈的中心点称作这个簇的中心（centroid），或平均值（mean或average)。这个点的坐标值就是这个簇中所有点x和

*y*坐标值的平均值。

本章，我们把聚类设想成一个几何问题，且还会运用几何学技巧来解释各种距离的度量方法，它们是聚类算法的核心。我们还会关注几个重要的距离测度方法以及它们与聚类的关系。这将有助于你理解文本文件的聚类与平面上点的聚类是何其相似。

在后面的章节里，我们会探讨一些数据聚类时的常用方法，以及它们在Mahout中的软件实现。在图书馆案例中采用的策略是合并书堆直到达到某个阈值。这个例子中的簇个数取决于这些数据；根据书的个数和阈值的不同，可能会有100个、20个簇，也可能只有1个簇。更为常见的策略是设定一个簇的目标数量，而不是使用阈值，然后就在这个限制下找到最佳的组合。稍后，我们详细诠释这种方法和一些变种。

7.2 项目相似性度量

聚类的关键在于寻找一个可以量化任意两个数据点之间相似性的函数。要注意我们在整本书中使用的两个术语：物品（item）和点（point），它们是可以互相替换的，都是指一个要被聚类的数据单元。

在*x*–*y*平面的例子中，对那些点的相似性度量（similarity metric）是指两个点之间的欧氏距离。图书馆的例子就没有如此明确的数学度量方法，而是完全依赖于图书管理员的智慧来判断书籍之间的相似性。这实际上并不满足我们的需求，因为我们要的是一个可以在计算机上实现的度量方法。

一种可能的度量方法是基于两本书的书名中相同词的数量。例如*Harry Potter: The Philosopher's Stone*和*Harry Potter: The Prisoner of Azkaban*两本书的书名中有3个共同词：Harry、Potter和The。然而用这种相似性度量方法就无法找到*The Lord of the Rings: The Two Towers*与Harry Potter系列的相似性，尽管它们是相似的书籍。你必须修改相似性的度量方法，使其能够顾及书籍本身的内容。你可以收集每本书的词频，如果这些书的词汇交集中有很多单词的词频数都比较接近，就可以判断这些书是相似的。

但遗憾的是，说起来容易做起来难。不仅因为这些书一般都有好几百页，还因为英语的语言特性本身也会让这种度量变得混乱。英语中最常用的一些词，如a、an和the，它们虽然总会在两本书中经常出现，但很难说明这两本书是相似的。

为了减少这些词的影响，你可以在计算的时候使用数值权重，给这些词较低的权重以减少它们对相似值的影响。你应该赋予在大多数书籍中都会出现的词较低的权重，而给那些只出现在少数几本书中的词较高的权重。对于会在某种特定书籍中才经常出现的词，你也可以给予它们较高的权重，因为这些词往往会明显地提示出书籍的内容，例如Harry Potter系列中的magic。

一旦你对一本书中的每个词都给出了权重，那么两本书之间的相似性就是在所有单词上的求和——某个特定词的出现频率乘以它们的权重。这是一个不错的度量方法，如果这两本书一样长的话。

如果一本书是300页，而另一本书是1000页，又会怎样呢？的确，一般来说，较长的书会有

较大的单词计数。你需要确保单词的权重与文章的长度相关。给任意一段话中的单词赋予权重，其本身就是一门学问。

TF-IDF（Term Frequency-Inverse Document Frequency，词项频率–逆文档频率）这个常用加权方法中包含了上述这些技巧以及许多其他技巧。本章会从平面上的点开始讨论，之后的几章会推进到对加权方法的理解，进而会深入到TF-IDF，探究对聚类质量的影响。下面，我们从一个Hello World聚类示例开始。

7.3 Hello World：运行一个简单的聚类示例

Mahout包含聚类的多种实现，如k-means、模糊k-means（fuzzy k-means）和canopy等。Hello World示例会运行k-means算法，而后续各章将审视Mahout中的其他算法及其实际应用。

7.3.1 生成输入数据

首先，让我们看一个简单的示例：对图7-1中二维空间中的点进行聚类。

我们首先创建一个点的列表以便聚类，Mahout的聚类算法的输入数据以一种特殊的二进制格式存储，称为`SequenceFile`，这是一种Hadoop常用的格式。输入数据写入多个`Vector`中，每个`Vector`代表一个点。

代码清单7-1 第一个聚类示例中的输入样本

```
(1,1)
(2,1)
(1,2)
(2,2)
(3,3)
(8,8)
(8,9)
(9,8)
(9,9)
```

看一下代码清单7-1中的输入数据样本。图7-2将之画在x–y平面上，可清晰地分出两个簇。一个簇包含平面上一个区域中的5个点，另一个包含另一个区域中的4个点。

为Mahout聚类算法输入数据的过程有三个步骤：预处理数据，使用数据生成向量，以及存为`SequenceFile`格式。对于点（point）而言，不需要做预处理，因为它们已经是二维平面的向量——你只需要将之转换为一个Mahout的`Vector`类。

当听到向量（vector）这个词时，你可能会想起高中物理课，向量在当时是指带有方向的箭头，而不是空间上孤立的点。在机器学习领域中，向量这个词指的是一个有序的数列，它究竟是一个点还是物理学的向量并不重要。向量有许多维度（这里是两维），每个维度都有一个数值。

提示 附录B解释了`Vector`接口及其实现的一些细节；必要时参考该附录可以更好地理解Mahout表达向量的方式。在我们的示例中，这些细节还不是很重要。

在后续各节中，我们会使用Mahout来聚类二维空间上的点。getPoints函数被用来将给定的输入点集合转换为RandomAccessSparseVector格式。一旦生成了这些向量，它们会被写为SequenceFile格式，以便Mahout中的聚类算法可以读懂。这个过程在writePointsToFile函数中实现。

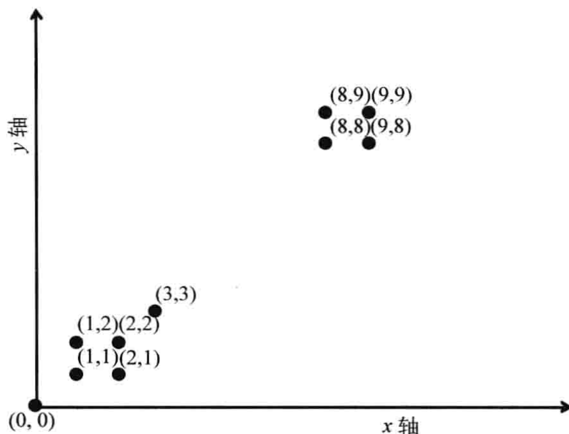


图7-2　在x–y平面显示代码清单7-1中的输入数据点

7.3.2　使用Mahout聚类

一旦准备好输入，就可以对数据点进行聚类了。这个示例中，我们使用k-means聚类算法，取如下输入参数。

- 包含输入向量的SequenceFile。
- 包含初始聚类中心点的SequenceFile。在本示例中，我们初始化两个簇，因此会有两个中心。
- 所用的相似性度量。这里我们使用EuclideanDistanceMeasure作为度量相似性的方法，本章稍后也会讨论其他的相似性度量方法。
- convergenceThreshold。如果在某次迭代中，簇中心的变化没有超过这个阈值，则不再进入下一次迭代。
- 迭代次数。
- 输入文件中使用的Vector实现。

万事俱备，只差初始的簇中心。要从9个点中生成两个簇，你需要两个点作为簇中心，如图7-3所示。尽量让这两个点是k-means算法所要寻找的两个簇中心的最佳猜测。

当然，你会发现这样的猜测不太准；它们通常会落入其中一个簇中。遗憾的是，在重要的案例中，没有办法预先知道簇的位置。估计簇中心的方法有很多——其中canopy聚类算法可以又快又好地给出估计值。

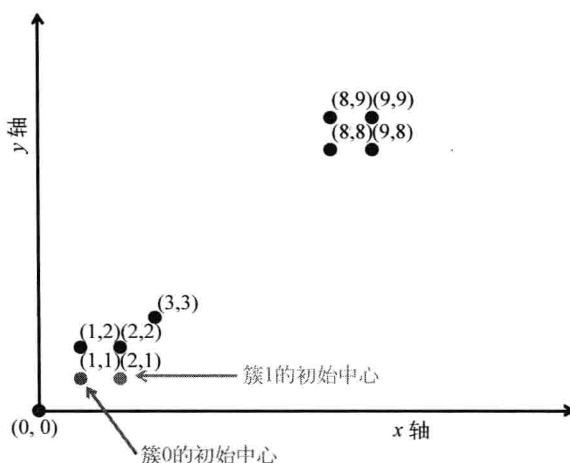


图7-3 指定初始簇是k-means聚类中的重要一步

即便估计的中心有误差，k-means算法也可以纠正它，在每次迭代的最后都会计算簇中所有点的平均中心，或称为中心点（centroid）。为了显示k-means的这种纠错性质，你可以选择两个靠得很近的中心点——(1, 1)和(2, 1)。你也可以尝试输入不同的中心值，来看一看k-means是如何在各种情况下使中心收敛的。

下面清单中的代码以in-memory模式使用Mahout的k-means，对平面上点的集合进行聚类。

代码清单7-2 Hello World聚类代码

```
public static final double[][] points = { {1, 1}, {2, 1}, {1, 2},
                                          {2, 2}, {3, 3}, {8, 8},
                                          {9, 8}, {8, 9}, {9, 9}};

public static void writePointsToFile(List<Vector> points,
                                     String fileName,
                                     FileSystem fs,
                                     Configuration conf) throws IOException {
  Path path = new Path(fileName);
  SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
      path, LongWritable.class, VectorWritable.class);
  long recNum = 0;
  VectorWritable vec = new VectorWritable();
  for (Vector point : points) {
    vec.set(point);
    writer.append(new LongWritable(recNum++), vec);
  }
  writer.close();
}

public static List<Vector> getPoints(double[][] raw) {
  List<Vector> points = new ArrayList<Vector>();
  for (int i = 0; i < raw.length; i++) {
    double[] fr = raw[i];
    Vector vec = new RandomAccessSparseVector(fr.length);
```

```
      vec.assign(fr);
      points.add(vec);
    }
    return points;
  }

  public static void main(String args[]) throws Exception {

    int k = 2;

    List<Vector> vectors = getPoints(points);

    File testData = new File("testdata");
    if (!testData.exists()) {
      testData.mkdir();
    }
    testData = new File("testdata/points");
    if (!testData.exists()) {
      testData.mkdir();
    }
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);
    writePointsToFile(vectors,
        "testdata/points/file1", fs, conf);

    Path path = new Path("testdata/clusters/part-00000");
    SequenceFile.Writer writer
        = new SequenceFile.Writer(
            fs, conf,       path, Text.class, Cluster.class);

    for (int i = 0; i < k; i++) {
      Vector vec = vectors.get(i);
      Cluster cluster = new Cluster(
          vec, i, new EuclideanDistanceMeasure());
      writer.append(new Text(cluster.getIdentifier()), cluster);
    }
    writer.close();

    KMeansDriver.run(conf, new Path("testdata/points"),
      new Path("testdata/clusters"),
      new Path("output"), new EuclideanDistanceMeasure(),
          0.001, 10, true, false);

    SequenceFile.Reader reader
        = new SequenceFile.Reader(fs,
            new Path("output/" + Cluster.CLUSTERED_POINTS_DIR
                + "/part-m-00000"), conf);

    IntWritable key = new IntWritable();
    WeightedVectorWritable value = new WeightedVectorWritable();
    while (reader.next(key, value)) {
      System.out.println(
          value.toString() + " belongs to cluster "
              + key.toString());
    }
    reader.close();
  }
```

指定所要形成的簇个数

为数据创建输入目录

写入初始中心点

运行k-means算法

读取输出，打印向量和簇ID

图7-4的流程图清晰地显示了代码清单7-2所做的工作。

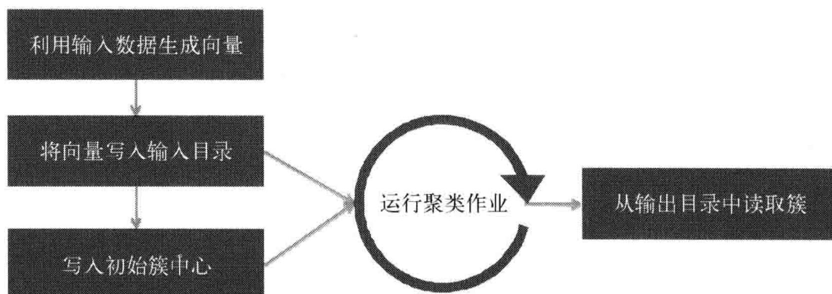


图7-4 最基础的聚类示例流程图

7.3.3 分析输出结果

使用你习惯的IDE或从命令行中编译并运行代码清单7-2中的代码：确保所有Mahout依赖的JAR文件都放在classpath中了。因为我们的数据集不大，你可以在几秒钟内得到如下的输出结果:

```
1.0: [1.000, 1.000] belongs to cluster 0
1.0: [2.000, 1.000] belongs to cluster 0
1.0: [1.000, 2.000] belongs to cluster 0
1.0: [2.000, 2.000] belongs to cluster 0
1.0: [3.000, 3.000] belongs to cluster 0
1.0: [8.000, 8.000] belongs to cluster 1
1.0: [9.000, 8.000] belongs to cluster 1
1.0: [8.000, 9.000] belongs to cluster 1
1.0: [9.000, 9.000] belongs to cluster 1
```

代码清单7-2用字符串标识符（string identifier）来唯一标记每个向量。这有助于你以后评估和重构这个簇。从图7-5可以看到，这个算法可以把簇的中心从(2, 1)调整到(8.5, 8.5)——簇1中所有点的中心。

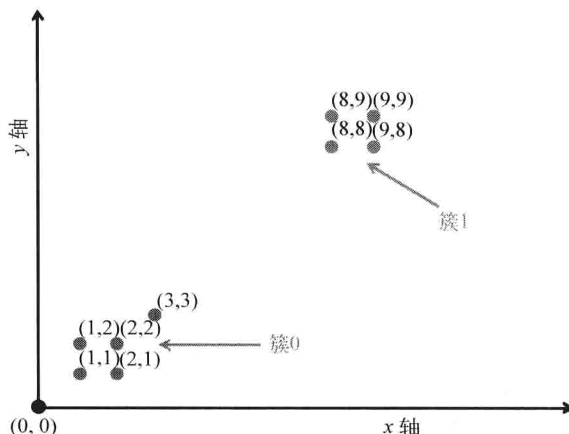


图7-5 一个最简单的k-means聚类算法的输出。即使初始的中心比较远，k-means算法也能够基于欧氏距离测度正确迭代，并对中心点进行纠正

在这个简单的示例中，Mahout将这些点快速而准确地分到两个集合中。实际场景中的数据不会如此简单。对于包含上百万个输入向量，而每个向量又包含上百万个维度的数据，快速地聚类就变得不那么轻松，会出现质量和性能上的问题。决定生成多少簇、或者选择哪种相似性度量将变得很困难。此外，还需要花大力气来调优性能以及评估簇的质量。要知道，实现一个完美的聚类是一项永无止境的工作。

7.4　探究距离测度

Mahout聚类的实现是可以灵活配置的，它足以胜任几乎所有的聚类问题，但是关键问题在于，到底哪种配置才是最优的？这里面的一个核心因素是距离测度的选择。

在之前的示例中，我们使用`EuclideanDistanceMeasure`来计算点和点之间的距离。虽然这被证明是生成簇的一种有效的度量手段，但在Mahout聚类包中还实现了其他的相似性度量。这些`DistanceMeasure`类恰如其名，它们根据各种对距离的定义来计算两个向量之间的距离。向量之间距离越短则越相似，反之亦然；相似性和距离在概念上是相互关联的。

7.4.1　欧氏距离测度

我们已经认识了欧氏距离（Euclidean distance），它是所有距离测度中最简单的。它最直观且符合我们通常对距离的理解。例如，给定平面上的两个点，欧氏距离测度可以通过使用一个标尺来计算出它们之间的距离。数学上，两个n维向量$(a_1, a_2,\cdots, a_n)$和$(b_1, b_2,\cdots, b_n)$之间的欧氏距离表示为：

No. 7

$$d=\sqrt{(a_1-b_1)^2+(a_2-b_2)^2+\cdots+(a_n-b_n)^2}$$

实现这个度量的Mahout类为`EuclideanDistanceMeasure`。

7.4.2　平方欧氏距离测度

正如名称所示，这种距离测度的值是欧氏距离的平方。对于n维向量$(a_1, a_2,\cdots, a_n)$和$(b_1, b_2, \cdots, b_n)$，其距离表示为：

$$d=(a_1-b_1)^2+(a_2-b_2)^2+\cdots+(a_n-b_n)^2$$

实现这个度量的Mahout类为`SquaredEuclideanDistanceMeasure`。

7.4.3　曼哈顿距离测度

不同于欧氏距离，在曼哈顿距离测度（Manhattan distance measure）中，两个点之间的距离是它们坐标差的绝对值之和。图7-6比较了在x-y平面上两个点的欧氏距离和曼哈顿距离。

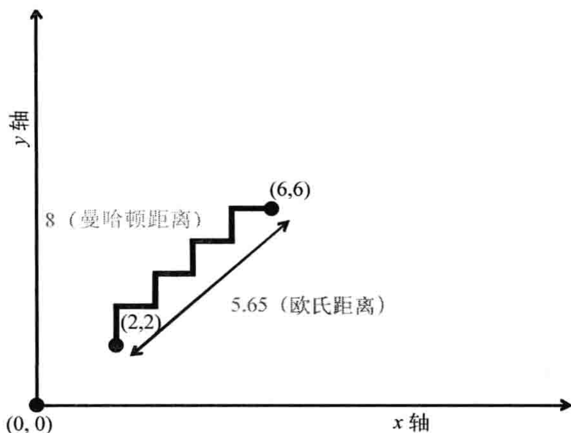


图7-6 欧氏距离和曼哈顿距离之间的区别。在(2,2)和(6,6)之间的欧氏距离为5.65，而曼哈顿距离为8.0

这个距离测度的名字取自呈网格状的曼哈顿街区。任何纽约人都知道，你不能直接穿越建筑从第2大道第2街区走到第6大道第6街区。实际步行距离为4个街区再加4个街区。数学上，两个n维向量$(a_1, a_2,\cdots, a_n)$和$(b_1, b_2, \cdots, b_n)$之间的曼哈顿距离表示为：

$$d = |a_1 - b_1| + |a_2 - b_2| + \cdots + |a_n - b_n|$$

实现这个度量的Mahout类为`ManhattanDistanceMeasure`。

7.4.4 余弦距离测度

余弦距离测度需要我们仍将这些点视为从原点指向它们的向量。这些向量之间形成了一个夹角θ，如图7-7所示。

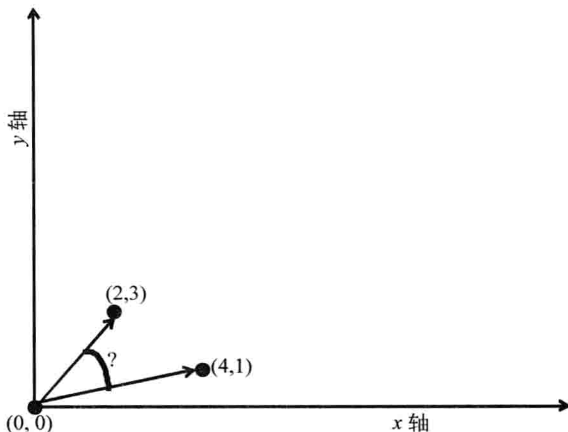


图7-7 向量(2,3)和(4,1)之间的余弦夹角（以原点为顶点）

当夹角较小时，这些向量都会指向大致相同的方向，因此这些点非常接近。当夹角非常小时，这个夹角的余弦接近于1，而随着角度变大，余弦值递减。余弦距离公式用1减去余弦值来得到一个合理的距离，这样0代表距离最近，而值越大则距离越远。

两个n维向量$(a_1, a_2, \cdots, a_n)$和$(b_1, b_2, \cdots, b_n)$之间的余弦距离公式为：

$$d = 1 - \frac{(a_1b_1 + a_2b_2 + \cdots + a_nb_n)}{\left(\sqrt{(a_1^2 + a_2^2 + \cdots + a_n^2)}\sqrt{(b_1^2 + b_2^2 + \cdots + b_n^2)}\right)}$$

注意，这种距离测度不考虑两个向量的长度，只关注于从原点到这两个点的方向。还需要注意余弦距离测度的范围是从0.0（两个向量方向相同）到2.0（两个向量方向相反）。

实现这个度量的Mahout类为`CosineDistanceMeasure`。

7.4.5　谷本距离测度

余弦距离测度忽略了向量的长度。这适用于某些数据集，但是在其他情况下可能会导致糟糕的聚类结果，因为向量的相对长度也会包含有价值的信息。

例如，考虑三个向量A (1.0, 1.0)、B (3.0, 3.0)和C (3.5, 3.5)。因为它们指向相同的方向，任意两个向量之间的余弦距离均为0.0。余弦距离无法看出B和C之间更为接近。欧氏距离测度可以很好地反映出这种差距，但它不会考虑向量之间的夹角——即它们方向相同的事实。有时，你也许希望找到一种可以同时表现夹角和距离的距离测度。

谷本距离测度（Tanimoto distance measure），也称为Jaccard距离测度，可以同时表现点与点之间的夹角和相对距离信息。两个n维向量$(a_1, a_2,\cdots, a_n)$和$(b_1, b_2,\cdots, b_n)$之间的谷本距离测度公式为：

$$d = 1 - \frac{(a_1b_1 + a_2b_2 + \cdots + a_nb_n)}{\sqrt{(a_1^2 + a_2^2 + \cdots + a_n^2)} + \sqrt{(b_1^2 + b_2^2 + \cdots + b_n^2)} - (a_1b_1 + a_2b_2 + \cdots + a_nb_n)}$$

实现这个度量的Mahout类为`TanimotoDistanceMeasure`。

7.4.6　加权距离测度

Mahout还提供了一个`WeightedDistanceMeasure`类，以及基于它的欧氏距离和曼哈顿距离测度实现。加权的距离测度是Mahout中的一种高级特性，允许对不同的维度加权从而提高或减小某些维度对距离测度值的影响。在`WeightedDistanceMeasure`中的权重需要以`Vector`格式序列化到一个文件中。

例如，当计算x–y平面点和点之间的距离时，假设你想让x坐标的影响是y坐标的两倍。当然，你可以让所有的x坐标值加倍。但要通过加权距离测度的方式实现，你就需要构建一个权重`Vector`，其第0个元素的值为`2.0`（用于x），第1个元素的值为`1.0`（用于y）。这会对不同距离测度产生不同的影响，但通常会让距离值对x值的变化更为敏感。

7.5 在简单示例上使用各种距离测度

分别使用欧氏、曼哈顿、余弦和谷本距离测度来运行前面那个最基本的k-means聚类程序，选择k=2（生成两个簇）。所得结果如表7-1所示。

表7-1 使用不同距离测度的聚类结果

距离测度	迭代次数	簇0中的向量[a]	簇1中的向量
EuclideanDistanceMeasure	3	0, 1, 2, 3, 4	5, 6, 7, 8
SquaredEuclideanDistanceMeasure	5	0, 1, 2, 3, 4	5, 6, 7, 8
ManhattanDistanceMeasure	3	0, 1, 2, 3, 4	5, 6, 7, 8
CosineDistanceMeasure	1	1	0, 2, 3, 4, 5, 6, 7, 8
TanimotoDistanceMeasure	3	0, 1, 2, 3, 4	5, 6, 7, 8

a. 它们是最基本聚类程序Hello World源代码中的向量名/ID

余弦距离测度聚类的结果有点让人费解。在图7-2中，你可以看到只有点(2, 1)和x轴的夹角大于45°。因此，聚类算法将小于和等于45°的所有其他点形成一个簇。这并不是说余弦距离测度是糟糕的——只是它对这个数据集不适用。比如，在文本聚类等领域中，它就表现得非常出色。

使用SquaredEuclideanDistanceMeasure增加了迭代的次数。这是因为在使用这个度量时，绝对距离值变得更大，而算法仍使用较小的convergenceThreshold值。因此，这使得达到收敛所需的迭代数增加了一倍。

以后的章节中，我们将看到更多的聚类方法，演示它们是如何适用于各种数据的，并解释如何使用各种距离测度来同时获得速度和质量上的优化。

7.6 小结

本章，我们介绍了聚类的思想。我们使用了一个直观的方法来聚类图书馆的藏书，接着使用二维空间上的点给出了聚类正式的定义。我们创建了平面上的一个很小的点集，并使用EuclideanDistanceMeasure运行了一个简单的k-means聚类示例。

接下来，我们讨论了Mahout中的各种距离测度。有了它们，我们重新运行了最初的示例并比较了使用这些距离测度后的聚类结果。

在详细地研究聚类算法之前，我们需要花一些时间了解Mahout中的基本概念——如何表示数据。这就是下面要讲的内容。

第 8 章

聚类数据的表示

8

本章内容

- 将数据表示为Vector
- 将文本文件转换为Vector形式
- 归一化数据表示

为了得到好的聚类结果，你就需要大致了解向量化技术：即把对象表示为Vector的过程。Vector是一种非常简单的数据表示方式，它可以帮助聚类算法理解对象，并计算它与其他对象之间的相似性。本章探讨各种把不同类型对象转换为Vector的方法。

在上一章，你已经对聚类有所了解。对书籍的聚类是基于它们在单词上的相似性，对二维平面中点的聚类是基于它们之间的距离。在现实中，聚类可以作用于任何类型的对象，只要你能区分出相似点和不同点即可。对图像的聚类可基于颜色、形状，或同时基于二者。而聚类照片时，你也许会试图将动物照片与人的照片区分开。你甚至可以通过动物的个体平均大小、重量和腿的个数自动地发现不同簇，从而对它们的种群进行聚类。

作为人类，我们可以聚类这些对象是因为我们理解它们，而我们“就是知道”什么是相似的而什么不是。不幸的是，计算机没有这种直觉，任何通过算法的聚类都要从对象的表示开始，这样我们才能让计算机可以读懂它们。

依据可衡量的特征或属性来考察对象已被证明是相当实用和灵活的。例如，个体大小和重量是两个显著的特征，它们可以帮助我们理解动物相似性的概念。每个对象（动物）在这些属性上都会有对应的数值。

我们希望把对象描述为值的集合，每个对象都关联到一个显著特征集合，或是一个维度集合——这听起来是不是很熟悉？我们又在描述一个向量。虽然你习惯于将向量视为空间中的箭头或点，但它们其实就是有序数列。因此，它们才可以很轻松地表示对象。

在上一章中，我们已经讨论了如何对向量进行聚类。但是，我们怎样才能在Mahout中表示向量？在此之前，我们又怎样由对象得到向量？这正是本章所讨论的内容。我们继续从数学概念出发讨论向量的由来。我们将解释如何把要聚类的数据转换成向量的形式，并通过封装使之能够为Mahout所理解。我们还将深入讨论文本数据以及一些重要的概念，如加权（weighting）和归一化（normalization）。最后，我们基于所有这些概念和向量化的路透社新闻数据集来使用Mahout库。

8.1 向量可视化

你可能在许多场景中见过向量这个词。在物理学中，一个向量代表一个力的大小和方向，或者一个移动物体（如一辆汽车）的速度。在数学中，一个向量只是空间中的一个点。这两种表现形式在概念上是非常相近的。

在二维空间，向量被表示为一个有序数列，每个维度有一个值，如(4, 3)。图8-1给出了这些表现形式。在处理二维问题时，我们常常将第一个维度称为x，而把第二个维度称为y，但这对Mahout而言并不重要。对我们而言，一个向量可以有2个、3个或10 000个维度。第一个是维度0，下一个是维度1，依此类推。

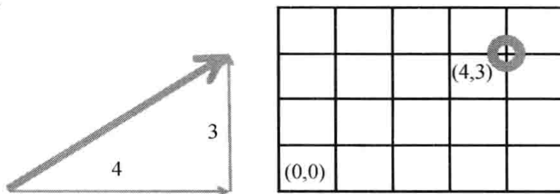


图8-1 在物理学中，一个向量可以看做是一条射线，有起点、方向和长度，并且表示了大小，如速度和加速度等。在几何或空间上，一个向量只是由每个维度的权重来表示的一个点。在默认情况下，向量的方向和大小由一条从原点(0, 0)出发的射线来表示

8.1.1 将数据转换为向量

在Mahout中，向量被实现为三个不同的类，每个类都是针对不同场景优化的：`DenseVector`、`RandomAccessSparseVector`和`SequentialAccessSparseVector`。

- `DenseVector`可被视为一个double型的数组，其大小为数据中的特征个数。因为不管数组的元素值是不是0，数组中所有元素都被预先分配了空间。我们称之为密集的（dense）。
- `RandomAccessSparseVector`被实现为`integer`型和`double`型之间的一个`HashMap`，只有非零元素被分配空间。因此，这类向量称为稀疏向量。
- `SequentialAccessSparseVector`实现为两个并列的数组，一个是`integer`型，另一个是`double`型。其中只保留了非零元素。与面向随机访问的`RandomAccessSparseVector`不同，它是为顺序读取而优化的。

附录B清晰地解释了它们之间的区别。

这三种实现允许你灵活地选择向量类，使这些类的性能特性能够适应数据特质、算法和数据访问方式。具体选择哪种实现依赖于算法。如果算法要对向量的值做许多随机插入和更新，就适合使用像`DenseVector`或`RandomAccessSparseVector`这样支持快速随机访问的实现。另一方面，而对于像k-means聚类这样反复计算向量大小的算法，`SequentialAccessSparseVector`实现的执行速度就会比`RandomAccessSparseVector`更快。

为了聚类对象，必须首先把它们转换为向量（将它们向量化）。每种类型的数据都有独特的向量化过程，但因为本书在这里讲述的是聚类，我们只谈论聚类的数据转换。我们希望读者此时已经可以容易接受把对象视为某种*n*维向量的方法。首先，对象需要被转换为一个向量，其维度数与对象的特征个数相同。让我们用一个示例来说明。

假设你想对一堆苹果进行聚类。它们有不同的形状、大小和颜色（红色、黄色和绿色），如图8-2所示。首先，你需要定义一种距离测度方法来说明两个苹果是相似的，只要它们仅在少数特征上有不同，且差别较少。相比于一个大的、卵型的、绿色的苹果，一个小的、圆的、绿色的苹果与一个小的、圆的、红色的苹果更相似。

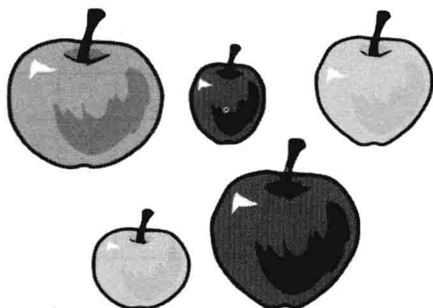


图8-2　不同大小和颜色的苹果需要转换成恰当的向量形式。诀窍在于找到如何将苹果的不同特征转化为十进制数值的方法

在向量化的过程中，首先需要将特征对应到维度上。让我们将重量（weight）作为特征（维度）0、颜色（color）为1而大小（size）为2。描述一个小的、圆的、红色的苹果的向量可表示为：

```
[0 => 100 gram, 1 => red, 2 => small]
```

但这个向量还没有数值，而这又是需要的。



对于维度0，你需要将重量表示为数字。这可以简单地用克（gram）或千克（kilogram）来测量。大小（维度2）未必能够等价于重量。我们都知道，绿色的苹果可能由于是新鲜的，而比红苹果的密度更高。此外还可以使用密度作为维度，只要我们有工具可以测量它。另一方面，大小可被置为人为观察到的数字：小苹果大小的值可以为1、中型苹果的大小为2，而大苹果的大小为3。

对于颜色（维度1）该怎么办呢？你可以为颜色分配任意的数字，比如红色=0.0，绿色=1.0，黄色=2.0。这是一种简单直观的表示方法；它适用于很多情况，但它没有反映出这样一个事实，即在可见光谱中，黄色是介于红色和绿色之间的颜色。我们可以通过更改映射关系来弥补这种不足，但也许更好的办法是直接用该颜色的波长（400~650 nm）。这样颜色就映射为一个有意义和客观的维度值。

用这些度量值作为苹果的属性，就可以为这些苹果生成向量，如表8-1所示。

表8-1 不同重量、大小和颜色的苹果数据集转换为向量

苹 果	重量（公斤）(0)	颜色 (1)	大小 (2)	向 量
小、圆、绿色	0.11	510	1	[0.11, 510, 1]
大、椭圆、红色	0.23	650	3	[0.23, 650, 3]
小、细长、红色	0.09	630	1	[0.09, 630, 1]
大、圆、黄色	0.25	590	3	[0.25, 590, 3]
中、椭圆、绿色	0.18	520	2	[0.18, 520, 2]

如果你不想基于颜色的相似度对苹果进行聚类，你还可以把颜色对应到不同的维度上。这就是说，红色为维度1，绿色为维度3和黄色为维度4。如果苹果是红色的，红色为值1和其他为0。然后可以用稀疏的格式存储这些向量，而距离测度只会考虑这些维度中非零值的存在，并将这些苹果中相同颜色的聚类在一起。

我们所选择的这种维度值映射方法有一个潜在的问题，即维度1中的值比其他维度大得多。如果我们采用一个简单的基于距离的度量标准来确定这些向量之间的相似性，颜色差异将主导最终的结果。比如，一个相对较小的10 nm的色差会相当于10倍大的形状差异，而在不同维度上加权可以解决这个问题。

权重的重要性将放在8.2节讨论，那时你会从文本文档中生成向量。在一个文档中，并非所有单词都对文档有相同的作用。权重技术可以帮助你强化更重要的词，而弱化那些相对并不重要的词。

8.1.2 准备Mahout所用的向量

有了将苹果编码为向量的方法之后，我们再看看如何准备Mahout算法所用的向量。首先，实例化一个Vector的实现，并把每个对象的值填充进去；再将所有Vector写入一个SequenceFile格式的文件，使其可被Mahout算法读取。SequenceFile是Hadoop库中的一种文件格式，它由一个键-值对序列组成。其中，键（key）须实现为Hadoop中的WritableComparable，值（value）须实现为Writable。在Hadoop中，它们等效于Java中的Comparable和Serializable接口。

例如，我们可以使用向量的名字或描述作为键，而向量本身作为值。Mahout的Vector类没有实现Writable接口，以避免它们和Hadoop直接耦合，但可以用VectorWritable类来封装一个Vector并使之为Writable。即Mahout中的向量可以使用VectorWritable类写入SequenceFile，如下所示。

代码清单8-1 为各种苹果生成向量

```
public static void main(String args[]) throws Exception {
  List<NamedVector> apples = new ArrayList<NamedVector>();

  NamedVector apple;
```

8

```
  apple = new NamedVector(
      new DenseVector(new double[] {0.11, 510, 1}),       将名字与向量关联
      "Small round green apple");
  apples.add(apple);
  apple = new NamedVector(
    new DenseVector(new double[] {0.23, 650, 3}),
      "Large oval red apple");
  apples.add(apple);
  apple = new NamedVector(
    new DenseVector(new double[] {0.09, 630, 1}),
      "Small elongated red apple");
  apples.add(apple);
  apple = new NamedVector(
    new DenseVector(new double[] {0.25, 590, 3}),
      "Large round yellow apple");
  apples.add(apple);
  apple = new NamedVector(
    new DenseVector(new double[] {0.18, 520, 2}),
      "Medium oval green apple");

  Configuration conf = new Configuration();
  FileSystem fs = FileSystem.get(conf);
  Path path = new Path("appledata/apples");
  SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
      path, Text.class, VectorWritable.class);
  VectorWritable vec = new VectorWritable();
  for (NamedVector vector : apples) {
    vec.set(vector);                                        序列化向量数据
    writer.append(new Text(vector.getName()), vec);
  }
  writer.close();

  SequenceFile.Reader reader = new SequenceFile.Reader(fs,
      new Path("appledata/apples"), conf);

  Text key = new Text();
  VectorWritable value = new VectorWritable();
  while (reader.next(key, value)) {                       反序列化向量数据

    System.out.println(key.toString() + " "
        + value.get().asFormatString());
  }
  reader.close();
}
```

选择一个对象的特征并把它们映射到数字的过程称为特征选择（feature selection）。将这些特征编码为向量的过程称为向量化（vectorization）。

基于对特征值的合理近似，任何类型的对象都可以被转换为向量形式，就像我们处理苹果的方法一样。现在让我们向量化一个特别有趣的对象类型：文本文档。

8.2　将文本文档表示为向量

数字形式的文本内容呈爆炸式增长。仅谷歌搜索引擎就索引了超过200亿个Web文档。这还

只是可以被公开抓取的信息的一部分。文本数据（公共的和私人的）的大小可能超越1个PB：就是1后面跟15个0。这对聚类和分类等机器学习算法来说是一个巨大的机会，即在非结构化的世界中找出一些结构和含义，而要做到这一点，首先需要学习文本向量化这门艺术。

VSM（Vector Space Model，向量空间模型）是向量化文本文档的常见方法。首先，考虑由一系列被向量化的文档中全部单词组成的集合。这个集合中的单词至少在任意一个文件中出现过一次。假定每个单词被分配一个编号，这个编号就是它在文档向量中所拥有的维度。

例如，如果单词horse被分配到向量的第39 905个索引位置，单词horse就对应文档向量的第39 905个维度。于是，文档的向量形式仅仅包含每个单词在文档中出现的次数，并且这个数值会被依次存储在各个单词所在的维度上。这些文档向量的维度会非常庞大。维度的最大可能数目就是向量的基数（cardinality）。因为可能出现的单词或词条（token）的个数无比巨大，文本向量通常被认为具有无限的维度。

对于一个单词而言，向量维度上的值通常是文档中单词出现的次数。这称为TF（Term Frequency，词频）权重。请注意，这些值也被称为这个字段上的权重（weight）。你可以看到有些参考文献使用了加权（weighting）这一说法，而不是值。单篇文档中出现的独立单词数目通常比在整个文档集合中的独立单词数目要少。其结果就是，这些高维度的文档向量相当稀疏。

在聚类中，我们经常基于距离测度来寻找两个文档之间的相似性。在典型的英文文档中，最频繁的词是a、an、the、who、what、are、is、was等。这类词称为停用词（stopword）。无论你使用任何距离测度计算两个文档向量之间的距离，你都会看到距离值会被这些频繁词的权重所左右。

这与之前的苹果与颜色问题是一样的。这种效果不是我们想要的，因为两个文档相似的主要原因是均出现了a、an和the这样的词。凭直觉，我们认为两份相似的文件应该谈论相似的主题，而标记一个主题的词通常是不常见的词语，像enzyme、legislation、Jordan等。这使得简单基于词频的权重并不适用于聚类，也不适用于其他需要计算文档相似度的应用。

幸运的是，我们可以使用非常简单但很有效的技巧来修复这些缺陷，从而改变加权方法，我们将在下面对此进行讨论。

8.2.1 使用TF-IDF改进加权

TF-IDF（Term Frequency–Inverse Document Frequency，词频–逆文档频率）加权被广泛用于改进简单的词频加权。改进之处在于增加了逆文档频率（IDF）部分，而不是简单地使用词频作为向量中的值，这个值会被乘以单词的文档频率的倒数。就是说，如果一个单词在所有文档中被使用的越频繁，那它对向量中的值的作用就会被抵消得越多。

为了解释这一点，我们假设一个文档中单词$w_1, w_2,\cdots, w_n$的频率为$f_1, f_2, \cdots, f_n$。单词w_i的词频（TF_i）为频率f_i。

为了计算逆文档频率，先计算每个单词的文档频率（DF）。文档频率是有这个单词出现的文档个数。单词在文档中出现的次数并不计入文档频率。那么，一个单词w_i的逆文档频率或IDF_i为：

$$\mathrm{IDF}_i=\frac{1}{\mathrm{DF}_i}$$

如果一个单词在文档集合中频繁出现，则其DF值大而IDF值小；这个IDF值会很小而使乘积后所得的权重值过小。在这种情况下，最好乘以一个常数来归一化IDF值。通常它被乘以文档个数（N），所以IDF的公式为：

$$\mathrm{IDF}_i=\frac{N}{\mathrm{DF}_i}$$

因此，在文档向量中单词w_i的权重W_i为：

$$W_i=\mathrm{TF}_i\cdot\mathrm{IDF}_i=\mathrm{TF}_i\cdot\frac{N}{\mathrm{DF}_i}$$

上述公式中的IDF值仍不理想，因为它掩盖了在最终的单词权重中TF的影响。为了解决这个问题，通常的做法是使用IDF值的对数：

$$\mathrm{IDF}_i=\log\frac{N}{\mathrm{DF}_i}$$

因此，对于单词w_i，TF-IDF权重W_i成为：

$$W_i=\mathrm{TF}_i\cdot\log\frac{N}{\mathrm{DF}_i}$$

也就是说，文档向量会把这个值放在单词i所对应的维度上。这就是经典的TF-IDF权重。停用词的权重小，而罕见的词的权重大。对重要的单词或主题词来说，通常有一个很大的TF值和比较大的IDF值，所以它们的乘积将成为更大的值，从而让这些词在所生成的向量中更加重要。

向量空间模型（VSM）有一个基本假设，即单词作为维度存在，因此是相互正交的。换句话说，VSM假定单词的出现是相互独立的，等同于点的x坐标完全独立于点的y坐标。凭直觉就知道这种假设在许多情况下都是错误的。例如，Cola这个词与Coca同时出现的概率会更高，所以这些单词并非完全独立的。因此，有一些其他的模型考虑了单词的依赖关系。

其中一个广为人知的技术是LSI（Latent Semantic Indexing，潜在语义索引），它检测可归并的维度并将它们合并成一个。由于维度减少了，聚类计算速度会更快。聚类质量也随之改善，因为现在我们会有一个非常好的属性能够出色地对文档对象进行分组。

在本书写作的时候，Mahout尚未实现LSI，但TF-IDF已被证明即使在独立性假设条件下也可以出色地工作。Mahout目前为单词依赖问题提供了一个解决方案，即通过使用一种称为搭配（collocation）或n-gram生成的方法，下面我们来看一下这个方法。

8.2.2　通过n-gram搭配词考察单词的依赖性

一个句子中的一组单词称为一个n-gram。一个单词可称为unigram，而像*Coca Cola*这样的两个单词可视为一个单位，并称为bigram。三个及以上单词的组合可称为trigram、4-gram、5-gram等。经典的TF-IDF权重假定单词的出现是独立于其他单词的，用这种方法创建的向量通常缺乏识

别文档关键特征的能力，因为这些特征可能是有依赖关系的。

为了克服这个问题，Mahout实现的技术能够识别出那些共现概率异常高的单词的组合，如*Martin Luther King Jr*或*Coca Cola*。在创建向量时，你可以不再把维度映射到单个词（unigram），而是同样简单地将其映射到bigram——或同时使用两种映射方法。于是，TF-IDF就可以像之前一样地发挥作用。

从一个由多个单词组成的句子中，你可以通过选择*n*个连续的单词来生成所有的n-gram。这个练习将生成许多n-gram，其中大部分并不是有意义的。例如，从句子"It was the best of times, it was the worst of times,"我们可以生成以下的bigram：

It was
was the
the best
best of
of times
times it
it was
was the
the worst
worst of
of times

其中有些组合很好，可用于生成文档向量（"the best,""the worst"），但有些却不够好（"was the"）。如果你将一个文档中的unigram和bigram合并，并使用TF-IDF来生成权重，结果就会让许多毫无意义的bigram占有较大的权重，因为它们有较大的IDF。这是极不可取的。

为了解决这个问题，Mahout利用一种称为对数似然（log-likelihood）的测试方法来考察*n*-gram，从而确定两个字在一起出现到底是偶然发生的，还是因为它们形成了一个有意义的单元。我们选择最有意义的，而排除最无意义的。在剩余的n-gram上就可以应用TF-IDF加权策略并生成向量。如此，在TF-IDF加权中就可以更合理地考量像*Coca Cola*这样有意义的bigram。

在Mahout中，使用`DictionaryVectorizer`类将文本文档通过TF-IDF加权和n-gram搭配词转换为向量。在下一节，你将看到如何从一个包含文件的目录开始，来创建TF-IDF加权向量。

8.3　从文档中生成向量

我们现在考察两个从文本文档生成向量的重要工具。第一个是`SequenceFilesFromDirectory`类，它将目录结构下的文本文档转换成以`SequenceFile`格式表示的中间文件。第二个是`SparseVectorsFromSequenceFiles`类，它使用基于n-gram的TF或TF-IDF加权将`SequenceFile`格式的文本文档转换为向量。`SequenceFile`格式的中间文件以文档ID为键；以

文档的文本内容为值。我们从一个文本文档目录开始，其中每个文件都包含一个完整的文档，之后，就可以使用这些类把文档转换到向量。

在这个例子中，我们使用Reuters-21578新闻数据集[1]。它在机器学习研究领域中被广泛使用。这些数据的收集和标记最初是由卡内基集团（Carnegie Group）和路透社（Reuters）在开发CONSTRUE文本分类系统的过程中完成的。Reuters-21578数据集分为22个文件，除最后一个文件（reut2-021.sgm）只包含578篇文档之外，其余每个文件均包含1000篇文档。

这些文件为SGML格式，类似于XML。我们可以为SGML文件创建一个解析器，把文档ID和文档文本写入SequenceFiles中，并使用SparseVectorsFromSequenceFiles将它们转换为向量。但更简便的方法是重用Lucene benchmark的JAR文件中给定的Reuters解析器。因为它就在Mahout的包中，你只需到Mahout源代码树的examples/directory下运行 org.apache.lucene.benchmark.utils.ExtractReuters类即可。

在此之前，从网站上下载Reuters数据集，并将其解压到examples/下的reuters/目录中。如下所示，在examples目录中运行Reuters的解压代码：

```
mvn -e -q exec:java
-Dexec.mainClass="org.apache.lucene.benchmark.utils.ExtractReuters"
-Dexec.args="reuters/ reuters-extracted/"
```

在解压后的目录中，运行SequenceFileFromDirectory类。你可以在Mahout的根目录下使用启动器（launcher）脚本完成相同的工作：

```
bin/mahout seqdirectory -c UTF-8
-i examples/reuters-extracted/ -o reuters-seqfiles
```

提示　你也许需要设置JAVA_HOME环境变量以运行这个启动器脚本

这会把Reuters的文章转换为SequenceFile格式。现在只差一步就可以将数据转换为向量。为了实现这一点，使用Mahout的启动器脚本运行SparseVectorsFromSequenceFiles类。

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-vectors -ow
```

提示　在Mahout中，使用-ow标志表示是否要覆盖输出文件夹。因为Mahout处理的数据集庞大，每个算法都需要花不少时间来生成结果。这个标志会防止意外删除需要花费数小时才能生成的输出。

Mahout启动器脚本中的seq2sparse命令从SequenceFile中读取Reuters的数据，并将基于词典的向量化程序所生成的向量写入输出目录中，该命令所用的默认选项在表8-2中简要列出。

---

① Reuters-21578测试数据集可在http://www.daviddlewis.com/resources/testcollections/reuters21578/中找到。直接的下载链接为http://www.daviddlewis.com/resources/testcollections/reuters21578/reuters21578.tar.gz。

表8-2 Mahout基于词典的向量化程序所用的重要标志及其默认值

选 项	标 志	描 述	默 认 值
覆盖（bool）	-ow	如果该标志被设置，则输出目录被覆盖。否则，当输出目录不存在时创建该目录，而当输出目录存在时，该作业失败并报错。默认为不设置	N/A
Lucene分析器名（String）	-a	所用分析器的类名	org.apache.lucene.analysis.standard.StandardAnalyzer
块大小（int）	-chunk	以MB为单位的块大小。对于大的文档集合（GB或TB级），你在向量化时无法将全部的词典装入内存，只有将词典分为特定大小的块，用多个步骤来执行向量化过程。建议你将这个块大小保持在Hadoop子节点上Java堆大小的80%，以避免向量化程序受到堆大小限制的影响	100
权重（String）	-wt	所用的加权机制：tf为基于词频的加权，而tfidf为基于TF-IDF的加权	tfidf
最小支持度（int）	-s	在整个集合中可放入词典文件的词的最小频率，低于该频率的词被忽略	2
最小文档频率（int）	-md	可放入词典文件的词所在文档的最小个数，低于该频率的词被忽略	1
最大文档频率（int）	-x	可放入词典文件的词所在文档的最大个数。这种机制用于去掉高频词（停用词）。所在文档比例大于该值的词都会被忽略	99
n-gram大小（int）	-ng	文档集合中选出的*n*-gram的最大长度	1
最小对数似然比（LLR，float）	-ml	这个标志仅当*n*-gram大于1时才生效。明显有意义的n-gram有很大的值，如1000；没什么意义的则值较低。虽然并无特定方法来选取该值，根据经验，LLR小于1.0的n-gram通常表示无意义	1.0
归一化（float）	-n	归一化值用在L_p空间。归一化的详细解释见8.4节。默认的策略是对权重不做归一化	0
reducer个数（int）	-nr	并行执行的reduce任务的个数。当基于目录的向量化程序运行在Hadoop集群上时，这个标志会生效。将它设置为集群的最大节点数会获得最高的性能。如果把这个值设置得比集群节点个数更大，会导致性能略为下降。通过阅读Hadoop文档，可以获得设置最优reducer个数的详细信息	1
生成顺序访问的稀疏向量（bool）	-seq	如果这个标志被设置，输出向量就被创建为SequentialAccessSparseVector。而默认情况下，基于目录的向量化程序创建的是RandomAccessSparseVector。前者在某些算法（如k-means和SVD）上可获得更高的性能，原因在于向量操作的连续访问特征。默认情况下，这个标志不被设置	N/A

看一下使用这些命令行命令所生成的目录：

```
$ ls reuters-vectors/
df-count/
dictionary.file-0
frequency.file-0
tfidf-vectors/
tf-vectors/
tokenized-documents/

wordcount/
```

在输出目录中，你会看到一个词典文件和几个目录。这个词典文件包含一个从词到它的整数型ID的映射。当你需要读取不同算法的输出结果时，这个文件就派上用场了，所以你需要保留这个词典文件。其他的文件夹是在向量化过程中所生成的中间文件夹，它们是由多个步骤（多个MapReduce作业）产生的。

第一步是文本文档符号化——它们被Lucene `StandardAnalyzer`拆分为单个的单词，并存储在tokenized-documents/文件夹中。第二步是字数统计——即*n*-gram生成（在本案例中仅计算unigrams）——迭代处理所有被符号化的文档，并生成重要单词的集合。第三步使用词频权重将符号化的文档转换为向量，进而生成TF向量。默认情况下，向量化程序使用TF-IDF，因此后面还有两个步骤：DF（Document-Frequency，文档频率）统计任务，以及TF-IDF向量生成。

TF-IDF加权的向量化文档可以在tfidf-vectors/文件夹下找到。对于大多数应用，你只需要这个文件夹和词典文件即可。

让我们重新审视Reuters `SequenceFiles`并使用非默认的值来生成一个向量数据集。所用的非默认标志值如下。

- ❑ `-a`　使用`org.apache.lucene.analysis.WhitespaceAnalyzer`基于单词之间的空白字符来标记单词。
- ❑ `-chunk`　使用200MB的块大小。这个值不会对Reuters数据产生任何影响，因为词典大小通常在1MB范围内。
- ❑ `-wt`　使用`tfidf`加权方法。
- ❑ `-s`　使用5作为最小支持度。
- ❑ `md`　使用3作为最小文档频率值。
- ❑ `-x`　使用90%作为最大文档频率百分比，以尽量去除高频词。
- ❑ `-ng`　使用2作为*n*-gram大小，以生成unigram和bigram。
- ❑ `-ml`　使用50作为对数似然比（LLR）的最小值，从而只保留有明显意义的bigram。
- ❑ `-seq`　设置`SequentialAccessSparseVectors`标志。

暂时不设置归一化标志（`-n`）。下一节我们再回来讨论这个标志。

在Mahout的启动器脚本中使用上述选项运行向量化程序。

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-vectors-bigram -ow
-a org.apache.lucene.analysis.WhitespaceAnalyzer
-chunk 200 -wt tfidf -s 5 -md 3 -x 90 -ng 2 -ml 50 -seq
```

这个向量化作业时生成的词典文件从654 KB增长到1.2 MB。虽然我们依据频率消除了更多的unigram，但是即便使用LLR阈值过滤，我们也会让bigram的个数几乎翻一倍。包含trigram之后，词典大小会增长到2 MB。至少在从bigram到trigram以及之后的过程中，它的大小还只是按线性增长的，这归功于基于LLR的过滤过程。否则，词典大小会呈指数增长。

至此，你已经可以尝试Mahout所提供的任何聚类算法了。在文本向量化中只有一个需要理解的重要概念了：归一化。我们下面来讨论它。

8.4 基于归一化改善向量的质量

归一化（normalization）在这里是一个清理边界情况的过程——带有异常特征的数据会导致结果出现不正常的偏差。例如，在使用某些距离测度计算文档之间的相似性时，总会有几个文档似乎与集合中所有其他的文档都相似，但仔细观察，你会发现这是因为这个文档非常大，并且它的向量有许多非零的维度，导致它和许多较小的文档都相似。因此，在计算相似性时，我们需要设法抵消向量大小不同所造成的影响。降低大向量的重要性，并提高较小向量的重要性的过程就称为归一化。

在Mahout中，归一化使用了在统计学中的p范数（p-norm）。例如，一个三维向量$[x, y, z]$中的p范数为：

$$\frac{x}{\left(|x|^p+|y|^p+|z|^p\right)^{1/p}},\frac{y}{\left(|x|^p+|y|^p+|z|^p\right)^{1/p}},\frac{z}{\left(|x|^p+|y|^p+|z|^p\right)^{1/p}}$$

这里，表达式$(|x|^p + |y|^p + |z|^p)^{1/p}$）可视为一个向量的范数（norm），我们就是让每个向量的值都除以这个数。参数p可以是大于0的任意值。向量的一范数（1-norm）或者曼哈顿范数（Manhattan norm）是这个向量除以所有维度的权重之和。

$$\frac{x}{|x|+|y|+|z|},\frac{y}{|x|+|y|+|z|},\frac{z}{|x|+|y|+|z|}$$

二范数（2-norm）或者欧氏范数（Euclidean norm）是这个向量除以它的幅值——我们可以将这个幅值习惯地理解为向量的长度。

$$\frac{x}{\sqrt{x^2+y^2+z^2}},\frac{y}{\sqrt{x^2+y^2+z^2}},\frac{z}{\sqrt{x^2+y^2+z^2}}$$

无穷范数（infinite norm）简单地将向量除以最大幅值维度的权重：

$$\frac{x}{\max\left(|x|,|y|,|z|\right)},\frac{y}{\max\left(|x|,|y|,|z|\right)},\frac{z}{\max\left(|x|,|y|,|z|\right)}$$

你选择的范数幂值（p）依赖于对该向量采取的是哪种操作。如果使用曼哈顿距离测度，一

范数通常会取得较好的结果。类似地，如果计算相似度使用的是欧氏距离测度的余弦值，向量二范数会得到较好的结果。总之，要得到最优结果，归一化应该关联到在相似性度量中所用的距离定义。

注意在p范数中的p可以是任意的有理数，因此3/4、5/3和7/5都是合法的归一化幂值。在词典向量化程序中，使用`-norm`标志设置这个幂值。INF的值代表一个无穷范数。生成这个用2来归一化的bigram向量与运行Mahout启动器一样简单，使用`seq2sparse`命令并把标识`-n`设为2：

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-normalized-bigram -ow
-a org.apache.lucene.analysis.WhitespaceAnalyzer
-chunk 200 -wt tfidf -s 5 -md 3 -x 90 -ng 2 -ml 50 -seq -n 2
```

归一化对聚类的质量有所提高。进一步改善聚类质量需要针对特定问题来设计距离测度和合适的算法。下一章，我们将呈现Mahout中的各种聚类算法。

8.5　小结

在本章中，你学习了聚类这类机器学习算法所使用的最重要的数据表示机制：`Vector`格式。在Mahout中有两种类型的`Vector`实现，稀疏（sparse）和密集（dense）向量。密集向量由`DenseVector`类实现；`RandomAccessSparseVector`是为满足应用快速随机读取而设计的一个稀疏实现，而`SequentialAccessSparseVector`则是为满足应用快速顺序读取而设计的。你可以根据你算法的访问模式来酌情选择。

你学习了如何将一个对象的重要特征映射为数值，进而生成代表不同对象类型的向量（在我们的示例中是苹果）。然后，就可以通过`SequenceFile`读写这些向量，它是Mahout中所有聚类算法都采用的格式。

文本文档在聚类中被频繁使用。使用向量空间模型（VSM）可以将文本文档表达为`Vector`。TF-IDF加权策略被证明是一种简单有效的方法，能够消除聚类过程中停用词所造成的负面影响。在经典TF-IDF加权策略中假设单词彼此独立，从而掩盖了文本中的一些重要特征，但是在Mahout中基于搭配的n-gram生成方法，通过使用对数似然比测试找出单词的明显分组，从而一定程度上解决了这个问题。Mahout基于词典的向量化程序可以轻松地将Retuers的新闻集合转化为向量。

最后，文本文档的长度对距离测度的质量有负面的影响。词典向量化程序所实现的p归一化方法通过除以向量的p范数来重新调整向量的权重，从而解决了这个问题。

使用Reuters的向量数据集，我们可以使用不同的技术来做聚类，它们各有利弊。在下一章，我们将探讨这些技术。

第 9 章

Mahout中的聚类算法

本章内容

- ❑ k-means聚类
- ❑ 使用canopy聚类生成簇的中心
- ❑ 模糊k-means聚类与狄利克雷聚类
- ❑ 聚类的一个变种，使用潜在狄利克雷分配对话题建模

现在，你已经知道输入数据如何表示为`Vector`，以及如何创建`SequenceFile`用作聚类算法的输入，你可以开始尝试Mahout提供的各种聚类算法了。Mahout包含了很多聚类算法，对于一个给定的数据集，某些算法适用，而另一些则不适用。k-means是一种通用的聚类算法，它可以容易地应用在大部分场合。它通俗易懂，而且可以很容易的在多台机器上并行执行。

因此，在了解各种聚类算法的细节之前，我们最好先通过k-means算法得到一些实践经验。在此基础上，更容易理解其他不太常见的技术有何缺陷与不足，并了解它们如何在特定场合更好的完成聚类。你将使用k-means算法对新闻文章进行聚类，并通过其他技术改善聚类质量。然后，你将学习如何利用canopy聚类推断k-means中的*k*值。有了这些知识，你将实现一个新闻聚合网站的聚类工作流，从而更好的认识如何用聚类解决真实世界中的问题。

熟悉了k-means之后，我们也会介绍它的一些缺点，以及其他特殊类型的聚类算法如何填补这些空白。我们会讨论模糊k-means和狄利克雷（Dirichlet）聚类在此类情况下的应用。最后，我们将介绍潜在LDA（Latent Drichlet Allcation，狄利克雷分配），一个与聚类非常相似的算法，但实现了一些更有趣的东西。

有很多需要介绍的东西，所以不要这里浪费时间了。我们现在就通过k-means算法进入聚类的世界。

9.1 k-means 聚类

k-means与聚类的关系，正如Vicks与止咳糖浆的关系一样。它是一个简单的算法，已经有50多年的历史。Stuart Lloyd于1957年首先提出了标准算法，并将其用于脉冲编码调制，但直到1982

年才发表[1]。它在众多科学领域中被广泛用作聚类算法。该算法要求用户设定聚类个数*k*作为输入参数。前面第7章中，我们对二维平面内的点进行聚类时已经使用过该算法。让我们来进一步了解算法的细节。

k-means算法有一个硬性限制，就是簇的个数*k*。你可能会质疑这一限制是否影响聚类效果，但这种担心是多余的。在其诞生的29年里，该算法已被证明能够广泛用于解决现实世界的问题。即使你估计的k值是次优的，聚类质量也不会受到太大影响。

假设你要对新闻报道进行聚类，以得到顶层类别，如政治、科学、体育等。对此，我们倾向于选择较小的*k*值，可能是10到20之间。如果需要细粒度的主题，则需要更大的*k*值，如50至100。假设你的数据库中有1 000 000篇新闻报道，需要按讨论的话题进行分组。这类相关话题的数量将远远小于整个语料库的大小——每个簇可能包含大约100篇报道。这意味着你需要使用大小在10 000左右的*k*值来生成这样一个分布。这个例子能够反映出聚类的可扩展性，而可扩展性正是Mahout的强项。

为使k-means得到较好的聚类质量，你需要首先估算*k*值。一个近似的方法是基于已有数据和需要的簇个数估计*k*值。在前面的例子中，我们有大约一百万篇新闻报道，如果平均每个话题有500篇相关报道，那么你就应该把聚类的*k*值设为2000（1 000 000/500）。

这是一种原始的估计簇个数的方法。然而，即使是这样粗略的估计，k-means算法也能得到令人满意的聚类结果。影响k-means聚类质量的决定性因素是所使用的距离测度的类型。在第7章中，我们提到了Mahout中各种各样的距离测度。我们将回顾这些距离测度方法，并在本章的例子中测试它们的效果。

9.1.1 关于 k-means 你需要了解的

让我们更进一步了解k-means算法。假设我们有*n*个点，需要聚到*k*个簇中。k-means算法首先从包含*k*个中心点的初始集合开始。随后，算法进行多次迭代处理并调整中心位置，直到达到最大迭代次数，或中心收敛于固定点不再移动。

图9-1展示的是一轮k-means迭代。实际算法是一系列这样的迭代。

此算法有两个步骤。第一步，找到距离各中心最近的数据点，并将这些数据点赋给特定的簇。第二步，使用各簇中所有点的坐标的均值更新中心位置。

这种两步算法是EM（Expectation Maximization，期望最大化）算法的一个经典例子。在EM算法中，两个步骤会重复执行直到收敛。第一步，称为E（Expectation，期望）步骤，寻找预期会与一个簇有关联的点。第二步，称为M（Maximization，最大化）步骤，利用E步骤获得的信息改善对簇中心的估计。对EM的详尽解释不在本书的讨论范围之内，但是有大量资源可供在线获取[2]。

---

① Stuart P. Lloyd, "Least Squares Quantization in pcm", IEEE Transactions on Information Theory, IT-28, 2: 129-137. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.1338.

② Frank Dellaert从下界最大化的角度解释了EM算法，"The Expectation Maximization Algorithm", http://www.cc.gatech.edu/~dellaert/em-paper.pdf。维基百科上也有关于此算法的词条：http://en.wikipedia.org/wiki/Expectation-maximization_algorithm。

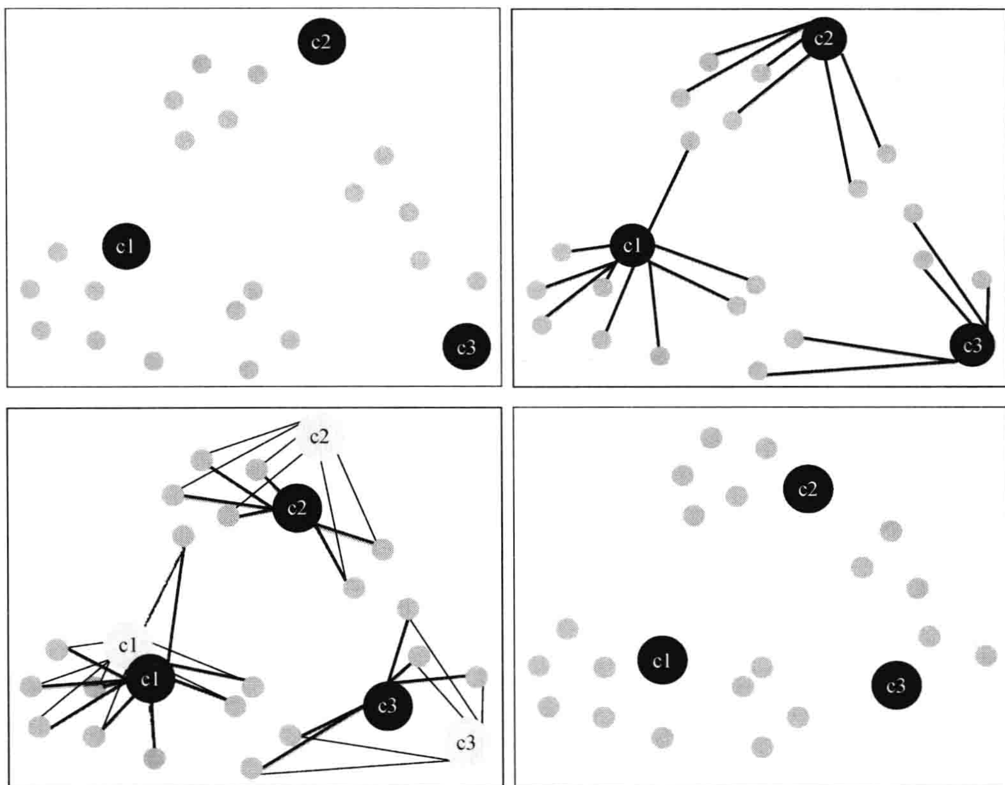


图9-1 k-means聚类实例。选择三个随机点用作聚类中心（左上），map阶段（右上）将每个点赋给离其最近的簇。在reduce阶段（左下），取相互关联的点的均值，作为新的簇中心位置，得到本轮迭代的最终布局（右下）。在每一轮迭代结束后，最终布局将被反馈给同样的循环过程，直到聚类中心的位置不再移动

在了解了k-means的技术之后，让我们看看在Mahout中非常重要的k-means类，并运行一个简单的聚类实例。

9.1.2 运行k-means聚类

k-means聚类算法可以通过`KMeansClusterer`或`KMeansDriver`类运行。前者以in-memory方式对数据点进行聚类，而后者则可以用于启动一个MapReduce作业来执行k-means。这两种方法不仅能像普通Java程序一样从磁盘上读写数据来运行，也可以在一个Apache Hadoop集群上执行，在分布式文件系统上读写数据。

在这个例子中，你将使用一个随机数据点生成器。它产生`Vector`形式的数据点，这些点以指定中心呈正态分布。你可使用Mahout中in-memory形式的k-means实现来对这些点进行聚类。

代码清单9-1中`generateSamples`函数可以使用如下输入参数，比如说是一个以(1, 1)为中

心、标准差为(2)、以及在中心附近呈正态分布的n(400)个随机点的集合。类似的，你还要生成另外两个点集，中心分别是(1, 0)和(0, 2)，相应的标准差分别为0.5和0.1。代码清单9-1使用如下参数运行KMeansClusterer：

- 输入点为List<Vector>格式；
- DistanceMeasure是EuclideanDistanceMeasure；
- 收敛阈值为0.01；
- 簇个数k为3；
- 初始中心由RandomPointsUtil选定，与第7章中的Hello World例子相同。

代码清单9-1　在内存中执行k-means聚类算法的示例。

```
private static void generateSamples(List<Vector> vectors, int num,
  double mx, double my, double sd) {
     for (int i = 0; i < num; i++) {
       vectors.add(new DenseVector(
         new double[] {
           UncommonDistributions.rNorm(mx, sd),
           UncommonDistributions.rNorm(my, sd)
         }
       ));
     }
  }
public static void main(String[] args) {
  List<Vector> sampleData = new ArrayList<Vector>();

  generateSamples(sampleData, 400, 1, 1, 3);
  generateSamples(sampleData, 300, 1, 0, 0.5);
  generateSamples(sampleData, 300, 0, 2, 0.1);

  int k = 3;

  List<Vector> randomPoints = RandomPointsUtil.chooseRandomPoints(
    sampleData, k);
     List<Cluster> clusters = new ArrayList<Cluster>();

     int clusterId = 0;
     for (Vector v : randomPoints) {
       clusters.add(new Cluster(v, clusterId++,
          new EuclideanDistanceMeasure()));
     }

     List<List<Cluster>> finalClusters
         = KMeansClusterer.clusterPoints(sampleData, clusters,
             new EuclideanDistanceMeasure(), 3, 0.01);
     for(Cluster cluster : finalClusters.get(
         finalClusters.size() - 1)) {
       System.out.println("Cluster id: " + cluster.getId()
           + " center: " +
           cluster.getCenter().asFormatString());
     }
  }
}
```

生成3个点集

运行KMeansClusterer

读取簇中心并打印

在Mahout的mahout-examples模块中有一个DisplayKMeans类，它是二维平面内算法可视化的一个有力工具。它能显示簇在每一轮迭代中如何移动。这也是一个展示KMeansClusterer如何做聚类的很好例子。以Java Swing应用程序的形式运行DisplayKMeans，并查看示例的输出，如图9-2所示。

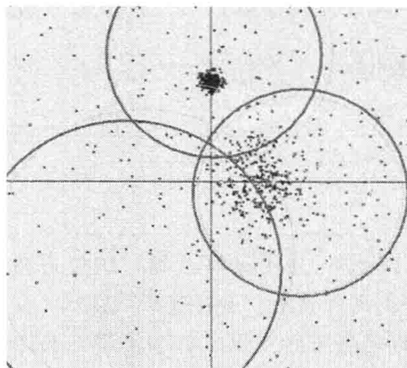


图9-2 在这个k-means聚类示例中，我们将k设为3，并试图对3个不同正态分布的数据点进行聚类。较细的线表示前一次迭代所估计的簇——你可以清晰的看到簇在移动

注意，k-means的in-memory聚类实现适用于Vector对象的列表。这个程序的内存用量取决于所有向量大小的总和。当向量为稀疏向量时，簇的大小要大于向量的大小，若为密集向量则二者大小相同。作为一条经验法则，内存需求量包括所有输入向量的大小之和，再加上k个簇中心的大小。但当数据量太大时，你就无法运行这一聚类实现。

而这正是MapReduce的长处。使用MapReduce架构，你可以将聚类算法分配到不同的机器上运行，每个mapper处理这些点的一个子集。Mapper作业将以流的形式读取输入数据点，并计算出距离这部分点最近的簇。

MapReduce版的k-means算法为Hadoop集群而设计，但是没有Hadoop时也能高效地运行。Mahout是在Hadoop代码之外编译的，这意味着你可以在没有Hadoop集群的情况下，直接在Java中运行同样的实现，并模拟Hadoop单台机器的情形。

1. 理解k-means聚类的MapReduce作业

在Mahout中，MapReduce版的k-means算法由KMeansDriver类实例化。该类只有一个入口——runJob方法。

你已经在第7章中见过k-means的实例。算法接受如下输入参数。

- Hadoop配置。
- 包含输入Vector的SequenceFile。
- 包含初始Cluster中心的SequenceFile。
- 用到的相似性度量。我们将使用EuclideanDistanceMeasure作为相似性度量，并在稍后试验其他度量方式。

- convergenceThreshold阈值。如果一次迭代中，中心移动量少于这个距离，则不再继续迭代，并终止聚类过程。
- 迭代数量。这是一个硬性限制；如果达到这个阈值，则聚类终止。

Mahout算法从不修改输入目录。因此你可以灵活地试验算法的各种不同参数。对于Java代码，你可以以如下代码清单中的方式调用入口函数，对文件系统中的数据进行聚类。

代码清单9-2　k-means聚类作业入口

```
KmeansDriver.runJob(hadoopConf,
     inputVectorFilesDirPath, clusterCenterFilesDirPath,
     outputDir, new EuclideanDistanceMeasure(),
     convergenceThreshold, numIterations, true, false);
```

提示　Mahout使用Hadoop的FileSystem类读写数据。这使得我们可以无缝对接到本地文件系统（通过java.io）和分布式文件系统，如HDFS和S3FS（使用Hadoop内部类）。这样一来，工作于本地文件系统上的代码同样可以工作于集群上的Hadoop文件系统，只要在环境变量中正确设置了Hadoop配置文件路径即可。在Mahout中，bin/mahout这个SHELL脚本会自动从$HADOOP_CONF环境变量寻找Hadoop配置文件。

我们将使用SparseVectorsFromSequenceFile工具（曾在前面第8章讨论过）将存放在SequenceFile中的文档转化为向量。因为k-means算法需要用户输入*k*个初始中心，MapReduce版本类似的需要你输入存放*k*个中心的文件系统路径。为了生成中心文件，你可以自定义一些逻辑来选定中心点，如我们在代码清单7-2的Hello World例子中所作的一样，或者你可以让Mahout随机生成*k*个中心，详细步骤如下。

2. 使用随机种子生成器运行k-means作业

下面我们按第8章（8.3节）介绍过的方式为Reuters-21578新闻集生成向量，然后在此基础上运行k-means聚类。在那一章中，新闻集被转化为Vector数据集，并使用TF-IDF度量作为权重。Reuters集包含很多话题类别，所以你可以把k设为20并观察k-means如何对集合中广泛的话题进行聚类。

要运行k-means聚类，我们的必选参数列表包含：

- Vector格式的Reuters数据集；
- 用于生成随机中心种子的RandomSeedGenerator；
- SquaredEuclideanDistanceMeasure；
- 较大的convergenceThreshold(1.0)，因为我们使用的是平方欧氏距离；
- maxIterations设为20；
- 簇个数*k*设为20。

如果我们通过DictionaryVectorizer将文本转化为向量时使用了多个reducer，SequenceFile格式的向量数据集通常会被分割为多个块。KMeansDriver假定输入目录中所有

的文件均为SequenceFile，并将它们全部读入。所以不必担心向量被分割到不同的块中，它们将由Mahout框架来处理。

包含初始中心的目录也是一样的。中心可能被写入到多个SequenceFile文件，Mahout会读取所有文件。对于实时插入数据的在线聚类系统，这一特性非常有用。系统建立一个新的独立文件块来写入数据，而不是附到已有文件的末尾，以免影响正在执行的算法。

警告 KMeansDriver接受一个初始簇中心目录作为参数。它仅在-k参数未设定的时候认为SequenceFile文件中包含了中心。若指定了-k参数，该类将删除此目录并向SequenceFile写入随机选择的*k*个点。

KMeansDriver还是对Reuters-21578新闻集进行k-means聚类的主入口点。在命令行中，以kmeans为程序名，在Mahout的exmples目录下执行Mahout启动器。KMeansDriver将使用RandomSeedGenerator随机选择*k*个簇中心并执行k-means聚类算法。

```
$ bin/mahout kmeans -i reuters-vectors/tfidf-vectors/ \
-c reuters-initial-clusters  \
-o reuters-kmeans-clusters \
-dm org.apache.mahout.common.distance.SquaredEuclideanDistanceMeasure \
-cd 1.0 -k 20 -x 20 -cl
```

我们使用Maven的Java执行插件来指定命令行参数并运行k-means聚类。-k 20参数指定了中心是由RandomSeedGenerator随机生成的，并且会写入到输入簇文件夹。距离测度SquaredEuclideanDistanceMeasure不需要显式设定，因为它是一个默认参数。

提示 你可以指定-h或--help命令行标志查看任何Mahout包的完整、详细的命令行标志和用法。

一旦命令开始执行，聚类迭代过程将一个接一个的运行。等待中心收敛需要点儿耐心。Hadoop监视程序会在一轮MapReduce的末尾打印计数器值，告诉你按照指定的阈值，有多少个中心已经收敛：

```
...
INFO: Counters: 14
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
INFO:   Clustering
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
INFO:     Converged Clusters=6
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
...
```

如果使用上述参数在内存中完成聚类，可能需要不到一分钟的时间。在同样的数据集上以MapReduce作业的方式执行同样的算法，则可能需要几分钟的时间。这里增加的时间来自Hadoop库的开销。在开始任何map或reduce任务之前，这个库需要做很多的检查，但一旦开始，Hadoop的mapper和reducer就会全速运行。在单机环境中，这个开销会降低系统的执行性能，但在集群上，这种延迟带来的不良影响会被并行计算所节省的时间抵消掉。

让我们回到运行k-means的控制台。在多个MapReduce作业之后，k-means簇收敛、聚类结束，然后点和簇的映射被写入输出文件夹。

提示　当你处理TB级的数据时，它们无法放入内存，MapReduce版的算法则具有较好的扩展性，它可以将数据存放在Hadoop分布式文件系统（HDFS）中并在大型集群上运行算法。如果你的数据量很小并且能够放入内存，那么就应该使用in-memory方式的实现。如果你的数据量大到无法放入内存，就应该使用MapReduce并考虑将计算放到Hadoop集群上完成。查看http://hadoop.apache.org/common/docs/r0.20.2/quickstart.html的Hadoop快速指南，可以找到更多关于在Linux机器上搭建伪分布式Hadoop集群的信息。

这一k-means聚类的实现在输出文件夹中建立了两类目录，clusters-*目录在每一轮迭代末尾生成：clusters-0目录在第1轮迭代之后生成，clusters-1目录在第2轮迭代之后生成，以此类推。这些目录包含了簇的信息：中心、标准差等。另一方面，clusteredPoints目录包含了从簇ID到文档ID的最终映射。这一数据是根据最后一轮MapReduce操作的输出生成的。

输出文件夹的目录列表与下面类似：

```
$ ls -l reuters-kmeans-clusters
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-0
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-1
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-2
...
drwxr-xr-x  4 user  5000  136 Feb 1 18:59 clusteredPoints
```

聚类完成之后，你需检查簇并观察它们是如何形成的。Mahout 提供了一个叫做`org.apache.mahout.utils.clustering.ClusterDumper`的功能，它可以读取任何聚类算法的输出，并显示各个簇的顶层条目，以及属于该簇的文档。运行如下命令执行`ClusterDumper`：[①]

```
$ bin/mahout clusterdump -dt sequencefile \
-d reuters-vectors/dictionary.file-* \
-s  reuters-kmeans-clusters/clusters-19 -b 10 -n 10
```

该程序需要词典文件作为输入。这是为了将特征ID或`Vector`的维度转化为词。

在最后一轮迭代的输出文件夹上执行`ClusterDumper`，会得到如下输出：

```
Id: 11736:
    Top Terms: debt, banks, brazil, bank, billion, he, payments, billion
     dlrs, interest, foreign
Id: 11235:
    Top Terms: amorphous, magnetic, metals, allied signal, 19.39, corrosion,
     allied, molecular, mode, electronic components
...
Id: 20073:
    Top Terms: ibm, computers, computer, att, personal, pc, operating system,
     intel, machines, dos
```

---

① Mahout-0.7中，需要通过-i指定聚类结果目录，并通过-o选项指定输出文件。——译者注

每次运行的结果会有所不同，原因在于*k*个中心是用随机种子生成器选择的，而最终结果很大程度上取决于这些中心。在前面的输出中，ID为11736的簇中靠前的单词为*bank*、*brazil*、*billion*、*debt*等。此簇中的大部分文章都是谈论这些话题的新闻。注意ID为20073的簇讨论的是涉及计算机及相关公司的话题（IBM、AT&T、PC等）。

正如你所看到的，使用`SquaredEuclideanDistanceMeasure`这个距离测度标准可以得到一个效果不错的聚类结果，但花了10次以上的迭代才得到最终结果。文本数据的特殊之处在于，两个内容相似的文档未必有相同的长度，而长度不同却话题相似的两个文档之间的欧氏距离会很大。也就是说，单词个数的差异对两篇文档欧氏距离的影响更大一些，公共词汇对两篇文档影响较小。为了更好地理解这一点，请重温7.4.1节中有关欧氏距离公式的实验。

这些因素导致欧氏距离不适用于文本文档。看看下面这个使用欧氏距离测度生成的簇：

```
Id: 20978:
    Top Terms: said, he, have, market, would, analysts, he said, from, which,
     has
```

这个簇实在是没有任何意义，特别是像*said*, *he*和*the*这些单词。要想真正在一个数据集上得到好的聚类结果，你需要尝试7.4节提及的Mahout中各种不同的距离测度，并比较它们处理你的数据集时所表现的性能。

我们已经知道余弦距离和谷本度量用于文本文档的效果很好，因为它们更依赖于公共词汇而受非公共词汇的影响较小。为验证这一点，我们将在Reuters数据集上进行实验，并将其输出与之前聚类的输出相比较。让我们来运行基于`CosineDistanceMeasure`的k-means：

```
$ bin/mahout kmeans -i reuters-vectors/tfidf-vectors/ \
-c reuters-initial-clusters  \
-o reuters-kmeans-clusters \
-dm org.apache.mahout.common.distance.CosineDistanceMeasure \
-cd 0.1 -k 20 -x 20 -cl
```

注意，本例中收敛阈值设为0.1，而不是默认值0.5，这是因为余弦距离的范围是0到1。程序运行时，有一点需要特别注意：由于余弦距离引入了额外的计算，聚类速度有所下降，但聚类过程在几次迭代之后就收敛了，而使用欧氏平方距离测度的聚类过程则需要10次以上的迭代。这清楚的表明余弦距离比欧氏距离更好的反映了文本文档的相似度。

聚类完成后，我们可以在结果上运行`ClusterDumper`，并查看各个簇中靠前的单词。下面是一些有趣的簇：

```
Id: 3475:name:
    Top Terms: iranian, iran, iraq, iraqi, news agency, agency, news, gulf,
     war, offensive
Id: 20861:name:
    Top Terms: crude, barrel, oil, postings, crude oil, 50 cts, effective,
     raises, bbl, cts
```

基于Mahout中k-means算法的实验，可以找到在特定聚类问题上`DistanceMeasure`和`convergenceThreshold`的最佳组合。可以在不同数据上进行尝试，并观察所得到的结果。你可以探究Mahout中的各种距离测度，或尝试自己的距离测度。虽然k-means可以在随机种子簇的基础上得到很好的结果，但最终的中心位置还是很依赖于它们的初始位置。

9

k-means算法是一种优化技术。给定初始条件，k-means试图把中心放到它们的最佳位置。但它是一种贪婪优化，这使得它只是寻找局部最优解。可能有其他中心位置也满足收敛性质，而且它们当中可能有些会比我们所得到的结果更好。我们可能永远也找不到完美的簇，但我们可以通过强有力的技术来逼近它。

9.1.3　通过canopy聚类寻找最佳k值

对于很多现实中的聚类问题，事先并不知道簇的个数，例如第7章中图书馆书籍分组的问题。有一类称为近似聚类算法的技术可以根据给定数据集估计簇的数量以及近似的中心位置。其中的一个算法称为canopy生成（canopy generation）算法。

默认情况下，Mahout中的k-means实现使用`RandomSeedGenerator`类生成包含*k*个向量的`SequenceFile`。尽管随机中心的生成速度很快，但无法保证为*k*个簇估计出较好的中心。中心估计极大地影响着k-means的运行时间。好的估计有助于算法更快地收敛，对数据的遍历次数也会更少。另外，最好能够根据数据自动确定簇的个数，但canopy算法仍然需要知道期望的簇大小，它才能找到接近该大小的簇个数。

1. 使用canopy生成算法来初始化k-means中心

canopy生成算法也被称为canopy聚类，是一种快速近似的聚类技术。它将输入数据点划分为一些重叠的簇，称为canopy。在这一上下文中，术语canopy指一组相近的点，或一个簇。canopy聚类基于两个距离阈值，试图估计出可能的簇中心（或canopy中心）。

canopy聚类的优势在于它得到簇的速度非常快，它只需遍历一次数据即可得到结果。这一优势也是它的弱点。该算法无法给出精准的簇结果。但它可以给出最优的簇数量，不需要像k-means那样预先指定簇数量*k*。

算法使用了一个快速的距离测度和两个距离阈值（T_1和T_2，其中$T_1>T_2$。它从一个包含若干点的数据集和一个空的canopy列表开始，然后迭代这些数据，并在迭代过程中生成canopy。在每一轮迭代中，它从数据集中移除一个点并将一个以该点为中心的canopy加入列表。然后遍历数据集中余下的数据点。对每一个点，它会计算其到列表中每个canopy的中心的距离。如果距离均小于T_1，则将其加入该canopy。若距离小于T_2，则将其移出数据集，以免在接下来的循环中用它建立新的canopy。重复上述过程，直到数据集为空。

这种方法可以防止紧邻一个现有canopy的点（距离小于T_2）成为新的canopy中心。我们不希望在一个现有canopy的附近生成一个冗余canopy。图9-3显示了使用此方法所创建的canopy。其中簇的形成仅仅依赖于距离阈值的选取。

2. 理解canopy生成算法

canopy生成算法通过`CanopyClusterer`或`CanopyDriver`类来执行。前者实现in-memory方式的聚类，而后者将其实现为MapReduce作业。这些作业可以像普通Java程序一样运行，读写磁盘上的数据。它们也能运行在Hadoop集群上，读写分布式文件系统上的数据。

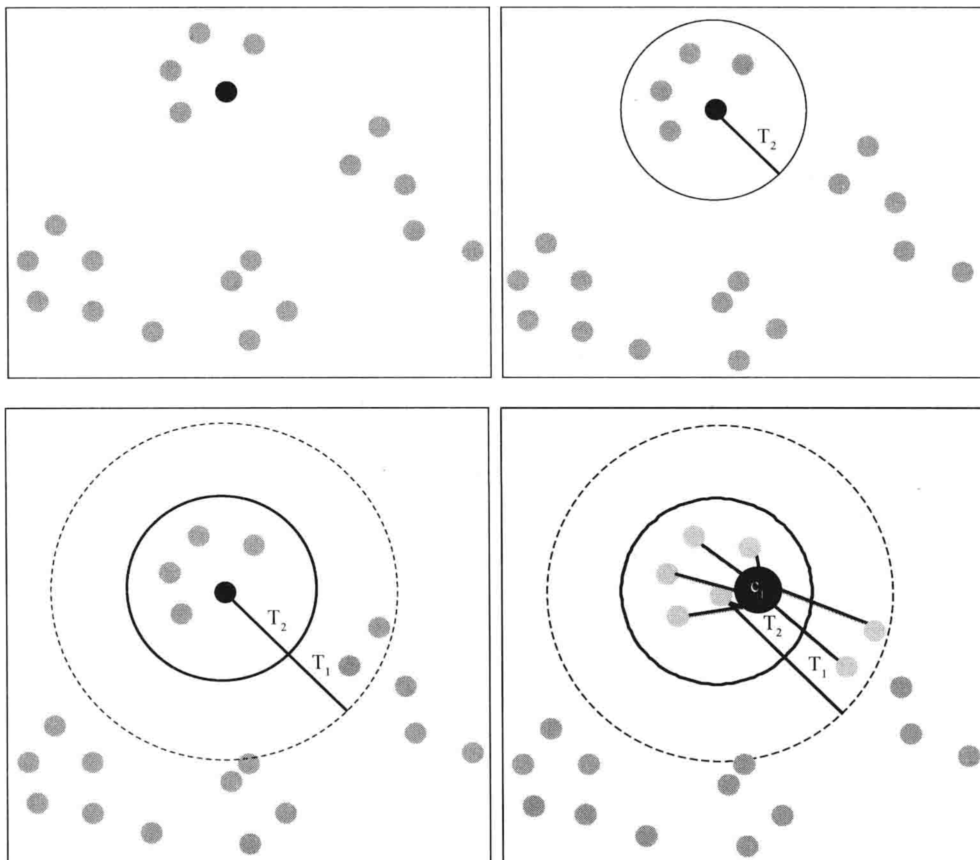


图9-3 canopy聚类：如果你从一个点开始（左上），并将其标记为一个canopy的一部分，距离T2以内的所有点（右上）被从数据集中删除，以免它们形成新的canopy。外圆内的点（左下）被放入同一canopy，但它们也可以属于其他canopy。这一分配过程是在一次mapper过程中完成的。Reducer计算中心均值（右下）并合并相近的canopy

我们这里所用的随机点发生器与前面在二维平面内所用的相同，也会生成服从正态分布的随机点。对于本例，我们将生成三维正态分布。代码清单9-3通过`CanopyClusterer`以in-memory方式运行canopy聚类，并使用如下参数：

- 输入向量为`List<Vector>`格式；
- `DistanceMeasure`为`EuclideanDistanceMeasure`；
- T_1的值为3.0；
- T_2的值为1.5。

代码清单9-3 以in-memory方式运行的canopy生成算法示例

```
public static void CanopyExample() {
```

```
List<Vector> sampleData = new ArrayList<Vector>();

generateSamples(sampleData, 400, 1, 1, 2);
generateSamples(sampleData, 300, 1, 0, 0.5);
generateSamples(sampleData, 300, 0, 2, 0.1);

List<Canopy> canopies = CanopyClusterer.createCanopies(
  sampleData, new EuclideanDistanceMeasure(), 3.0, 1.5);

for(Canopy canopy : canopies) {
  System.out.println("Canopy id: " + canopy.getId()
          + " center: " +
          canopy.getCenter().asFormatString());
 }
}
```

生成三个点集

运行CanopyClusterer

读取canopy中心并打印

在Mahoutmahout-examples模块中的DisplayCanopy类可显示二维平面内的点集，利用它可以显示出in-memory方式的CanopyClusterer是如何生成canopy的。DisplayCanopy的典型输出如图9-4所示。

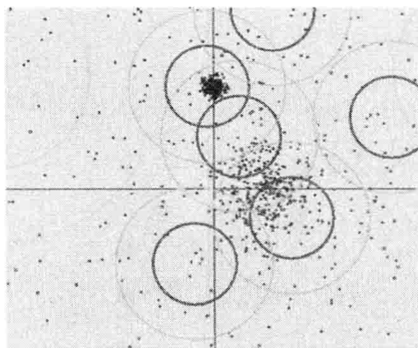


图9-4 使用DisplayCanopy类可视化一个in-memory方式的canopy生成示例，它包含在Mahout自带的示例程序中。我们使用参数T1=3.0和T2=1.5对随机生成的点进行聚类

canopy聚类不要求你指定簇中心的个数。中心个数的确定仅仅依赖于距离测度，T1和T2的选择。与k-means的实现类似，in-memory方式的canopy聚类作用于一个Vector对象的列表。如果数据集很大，这个算法就无法在单台机器上运行，而需要用MapReduce作业了。MapReduce版的canopy聚类实现使用了一点近似估算，所以对于同一个输入数据集来说，它生成的结果与in-memory版的结果有细微差别。当数据集很大时，这点区别是微不足道的。canopy聚类的输出很适合用来作为k-means的起始点，因为初始中心的准确性较之随机选择要高，所以能够改善聚类效果。

使用所生成的canopy，你可以将点赋给最近的canopy中心，理论上这就是对点进行聚类。我们称之为canopy聚类，而不是canopy生成。在Mahout中，CanopyDriver用于canopy中心的生成，而将runClustering参数设为true，即可实现对数据点进行聚类。

下面，你将在Reuters数据集上运行canopy生成，并确定*k*值。

3. 运行canopy生成算法来选择*k*个中心

现在我们来为Reuters的Vector数据集生成canopy中心。要生成中心，你需要先设定距离测度为EuclideanDistanceMeasure，并使用阈值t1=2000和t2=1500。注意，使用欧氏距离测度时，稀疏文档向量的距离会很大，所以要得到有意义的簇，需要把t1和t2的值设得大一些。

本例中选取的距离阈值（t1和t2）在Reuters数据集上生成了不到50个中心。我们在输入数据上多次运行CanopyDriver，就可以对阈值的选择有个估计。因为canopy聚类很快，使用多组不同的参数进行实验并观察结果要比像k-means这样耗时的技术快得多。

要在Reuters数据集上执行canopy生成的操作，只需通过Mahout启动器运行canopy程序，如下所示：

```
$ bin/mahout canopy -i reuters-vectors/tfidf-vectors \
-o reuters-canopy-centroids \
-dm org.apache.mahout.common.distance.EuclideanDistanceMeasure \
-t1 1500 -t2 2000
```

不到一分钟，CanopyDriver就会在输出文件夹中生成中心。你可以使用簇输出工具检查canopy中心，就如你在本章前面对k-means聚类结果所做的一样。

下面，我们将使用这个中心集合来改善k-means聚类。

4. 使用canopy中心改进k-means聚类

现在我们可以使用前一节中生成的canopy中心来运行k-means聚类算法了。为此，需要在KMeansDriver的簇参数（-c）中设置canopy聚类结果的输出文件夹，并去掉-k命令行参数。（注意：如果设定了-k标志，RandomSeedGenerator会覆盖canopy中心文件夹。）

我们将在k-means中使用TanimotoDistanceMeasure以得到簇：

```
$ bin/mahout kmeans -i reuters-vectors/tfidf-vectors \
-o reuters-kmeans-clusters \
-dm org.apache.mahout.common.distance.TanimotoDistanceMeasure  \
-c reuters-canopy-centroids/clusters-0 -cd 0.1 -ow -x 20 -cl
```

完成聚类后，使用ClusterDumper来检查簇，部分结果如下：

```
Id: 21523:name:
    Top Terms:
tones, wheat, grain, said, usda, corn, us, sugar, export, agriculture
Id: 21409:name:
    Top Terms:
stock, share, shares, shareholders, dividend, said, its, common, board,
     company
Id: 21155:name:
    Top Terms:
oil, effective, crude, raises, prices, barrel, price, cts, said, dlrs
Id: 19658:name:
    Top Terms:
drug, said, aids, inc, company, its, patent, test, products, food
Id: 21323:name:
    Top Terms:
       7-apr-1987, 11, 10, 12, 07, 09, 15, 16, 02, 17
```

注意最后一个簇。尽管其他簇看上去都是一些比较不错的话题，最后一个看起来却毫无意义。

不过，正是因为这些文档中同时出现了这些词条，聚类算法才将含有这些词条的文档归为一组。另外，像*its*和*said*这样的单词从语言角度来讲是没有意义的，但算法却不知道这一点。只要被赋予较大权重的向量特征能够很好地表现文档的特征，任何聚类算法都能得到好的聚类结果。

在8.3节和8.4节中，你已经看到TF-IDF和归一化如何将较高的权重赋给重要的特征，而将较低的权重赋给停用词，但即使这样，偶尔也会出现意想不到的簇。要避免此问题，一个快速而有效的方法就是将这些停用词从文档的`Vector`中删除。在下一个案例学习中，你将看到如何使用一个自定义的Lucene `Analyzer`类来解决此问题。

canopy聚类是一个很好的近似聚类技术，但它有内存限制。如果距离阈值很接近，就会产生太多的canopy，而这将增加mapper中的内存用量。当运行在一个很大的数据集上，又选择了不合适的阈值时，就可能会超出可用内存。下面你将看到，需要调优参数以适应这个数据集及其聚类问题。

下面我们将要看到的例子是为一个新闻网站创建聚类模块。我们之所以选取新闻网站的例子，是因为它代表了一种典型的动态系统，需要很精确地组织它的内容。聚类可以帮助解决与这种与内容系统相关的问题。

9.1.4　案例学习：使用k-means对新闻聚类

在这个案例中，我们将假设自己在管理一个虚构的新闻聚合网站，网站名为AllMyNews.com。访问网站的用户通过关键词搜索需要的内容。如果他们看到一篇有趣的文章，可以用文中的词汇搜索相关文章，也可以进入该文所属的新闻类别并浏览相关新闻。通常我们依靠人工编辑来寻找相关项，并协助对整个网站进行分类和设置交叉链接。但如果每天有数万条文章，人工干预的代价就太大了。下面我们讨论如何用聚类算法解决这一问题。

使用聚类算法，我们可以自动找到相关话题，并给予用户一个更好的浏览体验。本节中，你将使用k-means聚类实现这一功能。图9-5是该功能在实际应用中的一个例子。对于网站上的一则新闻报道，我们将为用户呈现一个相关新闻文章的列表。

图9-5　取自Google News网站的一个相关文章功能的示例。同一簇中类似报道的链接在底部以粗体字显示，而相关度靠前的文章在这些粗体字上面以链接的方式显示

对任何给定文章，我们都可以存储其所属的簇。当一个用户请求与他们正在阅读的文章相关的文章时，我们将选出该簇中的所有文章，并基于它们与给定文章的距离排序并呈现给用户。

对于新闻聚类系统来说，上面给出了一个很好的初始设计，但它并不完美。下面，我们列出

了一些实际应用中可能要面对的问题。

- 每一分钟都有文章到来，网站需要刷新它的簇和索引。
- 可能在同一时间内产生多个头条新闻，我们需要为它们建立不同的簇，这意味着每次发生这种情况时，我们都需要添加更多的中心。
- 文本内容的质量没有保证，因为数据有多个来源。我们需要在特征选择时有一种内容清理机制。

我们将首先实现一个高效的k-means聚类，离线地对新闻文章进行聚类。这里，离线这个词表示我们将把文档写入`SequenceFile`并以后台进程的形式启动聚类过程。我们不会深入探讨新闻数据如何存储。为简单起见，我们假设文档存储和检索模块不能简单地被替换为对数据库读/写的代码。

注意 在后续章节中，我们将修改这个案例，并引入Mahout中的一些高阶技术来解决与速度和质量相关的问题。最终，在第12章我们将展示一个可用的、调优的并且可扩展的聚类示例，用于一些现实中的数据集，并且可以适应不同的应用。

代码清单9-4展示了对`SequenceFile`中的新闻文章聚类的代码，代码清单9-5展示了一个自定义的Lucene `Analyzer`类，它将非字母字符从数据中删除。

9

代码清单9-4 使用canopy生成和k-means聚类对新闻进行聚类

```
public class NewsKMeansClustering {
  public static void main(String args[]) throws Exception {
    int minSupport = 2;
    int minDf = 5;
    int maxDFPercent = 95;
    int maxNGramSize = 2;
    int minLLRValue = 50;
    int reduceTasks = 1;
    int chunkSize = 200;
    int norm = 2;
    boolean sequentialAccessOutput = true;

    String inputDir = "inputDir";

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);

    String outputDir = "newsClusters";
    HadoopUtil.delete(new Path(outputDir));
    Path tokenizedPath = new Path(outputDir,
        DocumentProcessor.TOKENIZED_DOCUMENT_OUTPUT_FOLDER);
    MyAnalyzer analyzer = new MyAnalyzer();                    ← 自定义Lucene Analyzer
    DocumentProcessor.tokenizeDocuments(new Path(inputDir),
      analyzer.getClass().asSubclass(Analyzer.class),
      tokenizedPath, conf);                                    ← 文本词条化

    DictionaryVectorizer.createTermFrequencyVectors(tokenizedPath,
```

```
      new Path(outputDir), conf, minSupport, maxNGramSize, minLLRValue,
      2, true, reduceTasks,
      chunkSize, sequentialAccessOutput, false);
    TFIDFConverter.processTfIdf(
      new Path(outputDir ,
     DictionaryVectorizer.DOCUMENT_VECTOR_OUTPUT_FOLDER),
      new Path(outputDir), conf, chunkSize, minDf,
      maxDFPercent, norm, true, sequentialAccessOutput, false,
     reduceTasks);                                                计算TF-IDF向量
    Path vectorsFolder = new Path(outputDir, "tfidf-vectors");
    Path canopyCentroids = new Path(outputDir ,
                                "canopy-centroids");
    Path clusterOutput = new Path(outputDir , "clusters");     执行canopy中心生成

    CanopyDriver.run(vectorsFolder, canopyCentroids,
      new EuclideanDistanceMeasure(), 250, 120,
      false, false);                                           运行k-means算法
    KMeansDriver.run(conf, vectorsFolder,
      new Path(canopyCentroids, "clusters-0"),
      clusterOutput, new TanimotoDistanceMeasure(), 0.01,
      20, true, false);

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path(clusterOutput
            + Cluster.CLUSTERED_POINTS_DIR + "/part-00000"), conf);

    IntWritable key = new IntWritable();
    WeightedVectorWritable value = new WeightedVectorWritable();
    while (reader.next(key, value)) {
       System.out.println(key.toString() + " belongs to cluster "
       + value.toString());
    }                                                   读取Vector做聚类映射
    reader.close();
  }
}
```

代码清单9-5　一个自定义的用于过滤非字母字符的Lucene分析器

```
public class MyAnalyzer extends Analyzer {

  private final Pattern alphabets = Pattern.compile("[a-z]+");

  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
    TokenStream result = new StandardTokenizer(
        Version.LUCENE_CURRENT, reader);
    result = new StandardFilter(result);
    result = new LowerCaseFilter(result);
    result = new StopFilter(true, result,
                StandardAnalyzer.STOP_WORDS_SET);              使用Lucene过滤器

    TermAttribute termAtt =
       (TermAttribute) result.addAttribute(
           TermAttribute.class);
    StringBuilder buf = new StringBuilder();
    try {
```

```
    while (result.incrementToken()) {
      if (termAtt.termLength() < 3) continue;          // 过滤短词条
      String word = new String(
          termAtt.termBuffer(), 0, termAtt.termLength());
      Matcher m = alphabets.matcher(word);

      if (m.matches()) {                               // 过滤非字母词条
        buf.append(word).append(" ");
      }
    }
  } catch (IOException e) {
    e.printStackTrace();
  }

  return new WhiteSpaceTokenizer(new StringReader(buf.toString()));
 }
}
```

这个NewsKMeansClustering示例很简单。文档被存储在输入目录中。我们在此基础上创建仅包含字母字符的一元和二元组向量。使用生成的Vector作输入，我们运行canopy中心生成作业来创建k-means聚类算法的初始中心。最终，在k-means聚类结束后，我们读取输出并将其存入数据库。

下一节我们将在k-means的基础上更进一步，介绍Mahout中的其他聚类算法。

9.2 超越 k-means：聚类技术概览

k-means属于刚性的聚类。例如，一则谈论政治对生物技术影响的新闻报道，既可以归为政治类别，也可以归为生物技术类别，但不能同时归为这两个类别。既然我们希望优化相关文章这一特性，那可能就需要允许重叠或模糊的信息。我们也许还需要对数据点分布建模，这已经超出了k-means设计的初衷。

k-means只是众多聚类算法中的一种。下面我们介绍其他一些为不同目的而设计的聚类算法。

9.2.1 不同类型的聚类问题

回想一下，聚类仅仅是一个对事物分组的过程。要想在简单分组的基础上更进一步，你需要了解不同类型的聚类问题。这些问题及其解决方案主要分为四类：

- 排他性聚类；
- 有重叠聚类；
- 层次聚类；
- 概率聚类。

下面我们逐一介绍。

1. 排他性聚类

在排他性聚类中，一个物品只能属于一个类别。回忆第7章中讨论过的图书馆书籍聚类问题。我们可以简单地把一本像《哈利·波特》这样的书归入小说类书籍。因此，《哈利·波特》只属于小说类。k-means实现的就是这种排他性聚类，因此，如果聚类问题需要这一约束，k-means通

常都可以搞定。

2. 有重叠聚类

如果我们想要做的是非排他性聚类，即不仅把《哈利·波特》归入小说中，而且将其归入年轻人类别和玄幻类别。有重叠聚类算法，如模糊k-means等就很容易实现这一点。此外，模糊k-means能显示出对象与簇的相关程度。相对于年轻人类别，《哈利·波特》可能与玄幻更为相近。排他性聚类与有重叠聚类之间的区别在图9-6中给出。

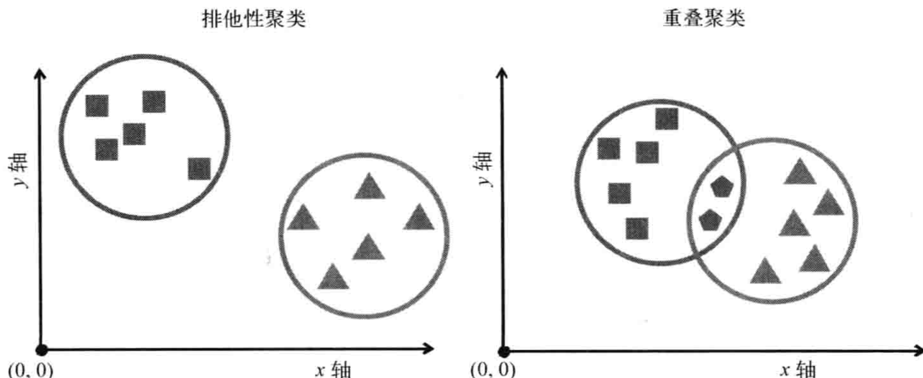


图9-6　排他性聚类与有重叠聚类在两个中心上的对比。对于前者，方块和三角形分别有自己的簇，且每个图形仅仅属于一个簇。而在有重叠聚类中，某些形状（如图中的五角形）可以同时属于两个不同的簇，因此它们同时是两个簇的组成部分

3. 层次聚类

现在，假设我们有两个簇分别代表不同类型的书籍，其中一个是玄幻，另一个是太空旅行。《哈利·波特》属于玄幻类书籍，但太空旅行和玄幻这两个簇都可以被看做是小说的子簇。因此，通过合并这两个簇，以及其他一些相似的簇，我们可以构建出一个小说簇。现在，小说和玄幻簇有了父子关系，如图9-7所示，因此称为*层次聚类*（hierarchical clustering）。

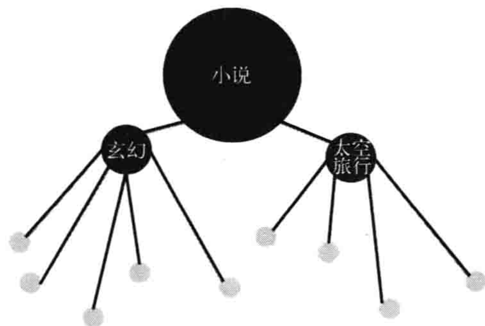


图9-7　层次聚类：一个较大的簇和两个较小的簇以树型结构组织在一起，构成一种层次化结构。回忆第7章中的例子——我们简单地依据相似度将书籍堆叠到一起时，便是完成了一个粗略的层次聚类

类似地，我们可以持续地合并簇，不断得到更大的簇。当进行到某种程度时，簇会变得过于庞大和宽泛，可能已经不再是有意义的分类。即便如此，这仍是一个十分有用的聚类方法：合并小的簇，直到得到满意的结果。在给定数据集上发掘这种系统性层次结构的方法称为层次聚类算法。

4. 概率聚类

一个概率模型通常是一个*n*维平面内一组点的分布或形状特征。有很多种适合不同数据模式的概率模型。概率聚类算法设法为数据集拟合出一个概率模型，通过调整模型参数来适应数据集。因为少有完全准确的拟合，所以这些算法通常给出一个百分比或概率值来表示概率模型对簇的拟合程度。

为了解释如何产生这种拟合，我们看一下图9-8中那个二维的例子。假设我们已知平面上的点分布在若干个椭圆形区域中，但我们不知道这些区域的中心、半径和轴线。我们可以选择一个椭圆形模型并试图将其拟合到数据上；并对每一个椭圆区域进行平移、拉伸或收缩，以得到最佳拟合结果。这称为基于模型的聚类。

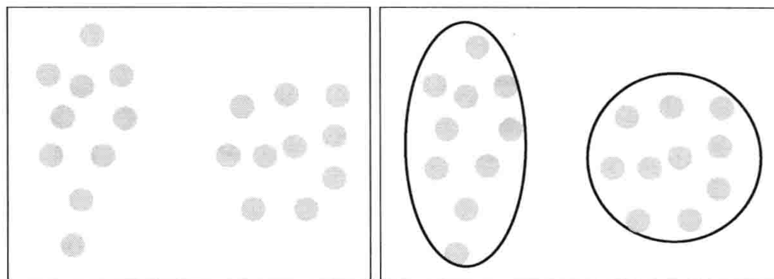


图9-8 概率聚类的简化图示。左图是初始点集。右图第一组点与一个细长的椭圆模型相匹配，而第二个则更为对称

此类方法的一个典型例子就是狄利克雷聚类算法，它根据用户提供的一个模型做拟合。我们将在9.4节看到这个聚类算法的实例。在这之前，你需要了解我们是如何根据不同聚类算法的策略对其分组的。

9.2.2 不同的聚类方法

不同聚类算法使用不同的策略。我们可以将其大致分为如下几类：

- 确定的中心个数；
- 自底向上的方法；
- 自顶向下的方法。

还有许多其他的聚类算法，具有独特的聚类策略，但你可能永远不会在Mahout中遇到它们，因为它们目前在大数据集上不具有可扩展性。这里，我们仅探讨上述三类算法。

1. 确定的中心个数

这些的聚类方法预先确定了簇的个数。簇的数量通常用字母*k*表示，这起源于此类方法中最

著名的k-means算法。基本思想是从k个簇中心开始，并不断对其进行修正，以更好地适应数据。一旦中心在数据上收敛，数据集里的点就被分配给距其最近的中心。

模糊k-means算法是另一个需要确定簇个数的例子。它不像k-means那样执行排他性聚类，模糊k-means执行的是可重叠聚类。

2. 自底向上的方法：通过组合，将点合并为簇

当你有一个n维点集时，可以做两件事：假设所有的点都属于同一个簇，并不断地将簇分解为更小的簇；或者假设每个点自己就是一个簇，并迭代地合并它们。前者是一种自顶向下的方法，后者则是一种自底向上的方法。

自底向上的聚类算法工作方式如下：算法从一个n维空间内的点集开始，寻找最接近的点对，并将它们合并为一个簇，如图9-9所示。这一合并仅在二者距离小于特定阈值时执行。否则，不对这些点做任何操作。这个根据距离合并簇的过程会不断重复，直到没有可合并的簇为止。

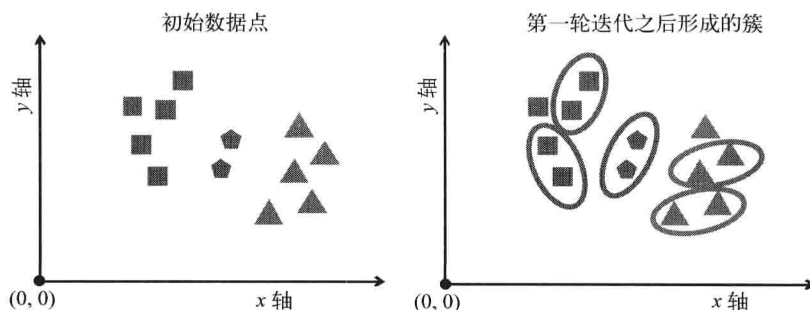


图9-9　自底向上的聚类：在每一轮迭代之后，相互靠近的簇会被合并而产生越来越大的簇，直到在给定的距离测度下没有可以合并的簇为止

3. 自顶向下的方法：拆分大簇

在自顶向下的方法中，你需要先将所有点都赋给同一个大的簇。然后你需要寻找最好的拆分方式来将这个大的簇一分为二，如图9-10所示。基于给定距离测度标准，反复划分这些簇，直到你得到有意义的簇为止。

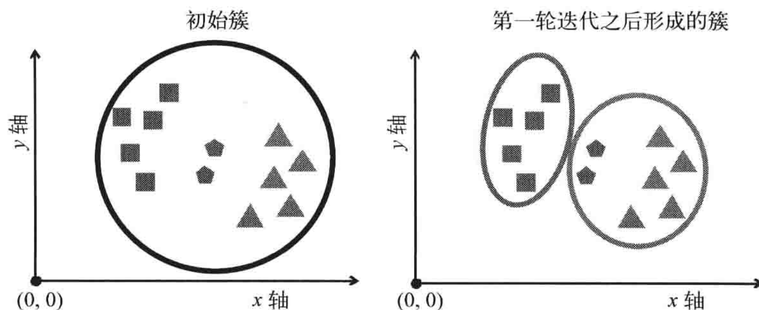


图9-10　自顶向下的聚类：在每一轮迭代中，寻找最优分割将簇一分为二，直至得到理想的聚类结果

虽然看起来很简单，但要找到一个*n*维数据点集合的最佳划分并不容易。此外，大部分此类算法都不能被直简化为MapReduce的形式，因此目前Mahout中并没有包含这些算法。

一个自顶向下方法的例子是谱聚类（spectral clustering）。在谱聚类中，我们寻找一条分界线或平面将数据划分为两个集合，使得两个集合的边界尽可能大。

4. 自顶向下和自底向上方法的优缺点

自顶向下和自底向上方法的诱人之处在于它们不需要用户输入簇的数目。这意味着对于数据点分布未知的数据集，这两类算法都能仅仅基于相似性度量给出聚类结果。这在很多应用中都是行之有效的。

然而，这些方法仍然处在研究当中，而且它们无法以MapReduce作业的方式运行。Mahout虽然没有实现这些方法，但提供了其他一些不需要指定簇数目的算法，它们能够以MapReduce作业的方式运行。

虽然没有支持MapReduce的层次聚类算法，但我们可以通过巧妙使用k-means、模糊k-means和狄利克雷聚类来对其进行模拟。要得到层次结构，可以从较小的簇个数（*k*）开始，并在反复聚类的过程中不断增大*k*值。另外，你也可以从大量中心开始，然后在聚类的过程中减小*k*值。这样，我们充分利用Mahout的可扩展性，模拟了层次聚类的行为。

下一节将详细讲解模糊k-means算法。我们用它来改善新闻网站AllMyNews.com的相关文章聚类。

9.3 模糊 k-means 聚类

顾名思义，模糊k-means聚类算法是k-means聚类的模糊形式。与k-means排他性聚类不同，模糊k-means尝试从数据集中生成有重叠的簇。在研究领域，这也叫做模糊c-means算法。你可以把它看成k-means的扩展。

k-means致力于寻找硬簇（一个点只属于一个簇），但是模糊k-means致力于寻找软簇。在一个软聚类算法中，任何点都能属于不止一个簇，而且该点到这些簇之间都有一定大小的吸引度。这种吸引度与该点到这个簇中心的距离成比例。同k-means一样，模糊k-means也适用于定义了距离测度的*n*维向量空间。

9.3.1 运行模糊k-means聚类

模糊k-means算法可通过`FuzzyKMeansClusterer`和`FuzzyKMeansDriver`两个类来使用。前者是一个in-memory的实现，后者则使用了MapReduce。

我们来看一个例子。仍然使用前面的随机点发生器来生成一些二维平面内分散的点。代码清单9-6用`FuzzyKMeansCluster`展示了in-memory形式的实现，所用参数如下：

- 输入`Vector`数据为`List<Vector>`格式；
- `DistanceMeasure`是`EuclideanDistanceMeasure`；
- 收敛阈值为0.01；

- 簇个数k为3；
- 模糊参数（该参数的详细解释见9.3.2节）m为3。

代码清单9-6　in-memory形式的模糊k-means聚类示例

```
public static void FuzzyKMeansExample() {
  List<Vector> sampleData = new ArrayList<Vector>();
                                                          生成3个点集
  generateSamples(sampleData, 400, 1, 1, 3);
  generateSamples(sampleData, 300, 1, 0, 0.5);
  generateSamples(sampleData, 300, 0, 2, 0.1);

   int k = 3;
  List<Vector> randomPoints
     = RandomPointsUtil.chooseRandomPoints(sampleData, k);
     List<SoftCluster> clusters = new ArrayList<SoftCluster>();

     int clusterId = 0;
  for (Vector v : randomPoints) {
    clusters.add(new SoftCluster(v, clusterId++,
        new EuclideanDistanceMeasure()));
  }

  List<List<SoftCluster>> finalClusters
      = FuzzyKMeansClusterer.clusterPoints(sampleData,
          clusters, new EuclideanDistanceMeasure(),
          0.01, 3, 10);
  for(SoftCluster cluster : finalClusters.get(           运行FuzzyKMeansClusterer
      finalClusters.size() - 1)) {
 System.out.println("Fuzzy Cluster id: "
     + cluster.getId()
     + " center: " + cluster.getCenter().asFormatString());
  }                                                       读取模糊簇的中心

}
```

Mahout的`mahout-examples`模块中有一个`DisplayFuzzyKMeans`类，它是一个很好的工具，可以在二维空间中将该算法可视化。`DisplayFuzzyKMeans`是一个Java Swing应用程序，它产生图9-11所示的输出。

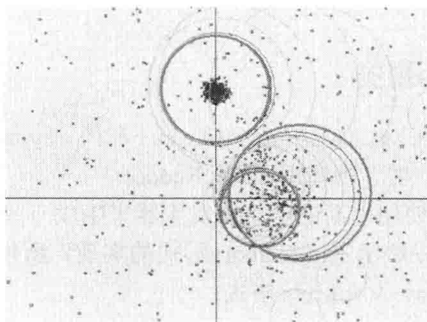


图9-11　模糊k-means聚类：这些簇看上去是互相重叠的，重叠程度由模糊因子决定

模糊k-means的MapReduce实现

Mahout中，模糊k-means的MapReduce实现与k-means类似。下面是一些主要参数。

- `Vector`格式的输入数据集。
- 用来生成*k*个初始簇的`RandomSeedGenerator`。
- 距离测度：这里使用`SquaredEuclideanDistanceMeasure`。
- 很大的收敛阈值`convergenceThreshold`，例如`-cd 1.0`，因为我们准备用距离的平方值。
- `maxIterations`值；我们准备用默认值`-x 10`。
- 归一化系数，或称模糊因子，该值大于 `-m 1.0`；我们会在9.3.2节详细描述该参数。

可以使用Mahout的启动器运行`fkmeans`，从而在输入数据上做模糊k-means聚类:

```
$ bin/mahout fkmeans \
-i reuters-vectors/tfidf-vectors/ -c reuters-fkmeans-centroids \
-o reuters-fkmeans-clusters -cd 1.0 -k 21 -m 2 -ow -x 10\
-dm org.apache.mahout.common.distance.SquaredEuclideanDistanceMeasure
```

与k-means一样，如果设定了簇个数（`-k`）标志，`FuzzyKMeansDriver`会自动运行`RandomSeedGenerator`。生成了随机中心之后，模糊k-means聚类会用它们作为*k*个初始中心。该算法在输入数据集上执行多次迭代，直到中心收敛，每轮迭代都会在cluster-文件夹中创建新的输出文件。最后，它运行另一个作业，依据距离参数和模糊参数（`-m`），算出每一个点属于各个簇的概率。

在详细讨论模糊参数之前，我们最好先用`ClusterDumper`工具看看各个簇。`ClusterDumper`展示了每个簇中心的前几个词。为了获得点到簇的映射关系，你需要读取clusteredPoints/ 文件夹下的`SequenceFile`。这个序列文件中的每个条目都有一个键，它是向量的标识符；还包括一个值，这个值是一个簇中心的列表，每个簇中心还对应了一个数字，这个数字表示这个点符合这个中心的程度。

9.3.2 多模糊会过度吗

模糊k-means有一个参数*m*，叫做模糊因子。像k-means一样，模糊k-means循环地处理数据集，但不是把向量分配到最近的中心，而是计算每个点到每个簇的关联程度（degree of association）。

假设有一个向量V，到*k*个簇中心的距离分别为$d_1, d_2, \cdots, d_k$。向量（V）到第一个簇（C_1）的关联度（u_1）计算如下：

$$u_1 = \frac{1}{\left(\frac{d_1}{d_1}\right)^{\frac{2}{m-1}} + \left(\frac{d_1}{d_2}\right)^{\frac{2}{m-1}} + \cdots + \left(\frac{d_1}{d_k}\right)^{\frac{2}{m-1}}}$$

类似的，把分母表达式中的分子d_1替换为d_2，d_3等，可以计算出该向量与其他簇的关联度。从表达式中可以很明显的看出*m*应该大于1，否则分母就会是0，导致出错。

如果你把m设为2，则对于每个点来说，所有关联度的和为1。从另一个角度来看，如果m无限接近1，比如1.000 001，越接近该向量的簇中心，就越会得到更高的权重。如果m趋近于1，模糊k-means算法就很接近k-means算法了。如果m增大，会使得算法的模糊度增大，因而产生更多重叠。

模糊k-means算法比标准k-means算法收敛得更好、更快。

9.3.3 案例学习：用模糊k-means对新闻进行聚类

如果允许簇之间有部分重叠，那么相关文章的功能显然会更丰富。重叠的分值有助于我们获得相关文章和簇的关联性，进而对它们进行排序。下面，我们修改9.1.4节中的学习示例，使用模糊k-means算法，并查看模糊簇的成员信息。

代码清单9-7　基于模糊k-means算法的新闻聚类

```
public class NewsFuzzyKMeansClustering {
  public static void main(String args[]) throws Exception {
    ...
    String vectorsFolder = outputDir + "/tfidf-vectors";
    String canopyCentroids = outputDir + "/canopy-centroids";
    String clusterOutput = outputDir + "/clusters/";

    CanopyDriver.run(conf, new Path(vectorsFolder), new Path(
        canopyCentroids), new ManhattanDistanceMeasure(), 3000.0,
      2000.0, false, false);                                   <-- 运行canopy生成作业

    FuzzyKMeansDriver.run(conf, new Path(vectorsFolder), new Path(
        canopyCentroids, "clusters-0"), new Path(clusterOutput),
      new TanimotoDistanceMeasure(), 0.01, 20, 2.0f, true, true, 0.0,
      false);                                                  <-- 运行模糊k-means聚类

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path(clusterOutput + Cluster.CLUSTERED_POINTS_DIR
                 + "/part-m-00000"), conf);

    IntWritable key = new IntWritable();
    WeightedVectorWritable value = new WeightedVectorWritable();
    while (reader.next(key, value)) {                          <-- 从输出中读取映射关系
      System.out.println("Cluster: " + key.toString()
        + " " + value.getVector().asFormatString());          <-- 打印簇和概率
      // 编写代码以保存簇映射到数据库
    }
    reader.close();
  }
}
```

模糊k-means算法给我们提供了一种改善相关文章代码的途径。现在我们可以知道一个点在多大程度上与一个簇相关。有了这个信息，我们就能找到最相关的簇，并且根据相关程度确定相关文章的带权分值。我们用这种方式避免了排他性聚类过于严格的限制，并且可以为处于簇边缘的文档识别出更好的相关文章。

9.4 基于模型的聚类

在本章中，聚类算法的复杂性逐渐提高。前面我们从k-means入手了解了聚类算法。接着讨论了模糊k-means产生的部分从属关系。我们还介绍了如何通过中心点生成算法来优化聚类，如canopy聚类。

关于这些聚类算法，我们还想知道些什么呢？为了更好地认识数据内在的组织结构，我们需要一个与前面完全不同的方法——基于模型的聚类。

在我们介绍基于模型的聚类算法之前，我们要先了解一下k-means和其他相关算法存在的问题。这一节我们要使用狄利克雷过程聚类（Dirichlet process clustering）来解决这些问题。但是，为了理解它，你首先要知道它解决的是什么问题，我们会把它和其他你之前看到的算法做一些比较。下面我们就看看什么是k-means算法所不能做的，然后了解狄利克雷过程聚类如何解决这个问题。

9.4.1 k-means的不足

假定你要把一个数据集聚类成k个簇。现在你已经知道如何运行k-means并快速得到这些簇。因为k-mean可以基于线性距离轻松地划分簇，它的性能一般都不错。

如果你知道这些簇都满足正态分布，并且相互混合与重叠在一起，该怎么办呢？这样的话你最好是使用模糊k-means聚类。

如果各个簇不是正态分布呢？如果这些簇呈卵形分布呢？无论k-means还是模糊k-means都无法利用这个信息来改进聚类过程。在我们探讨怎么解决这个问题之前，先来看一个例子，在这个例子中k-means聚类无法描述数据的简单分布。

9

1. 非对称正态分布

你将在一个按非对称正态分布生成的点集上运行k-means聚类。这意味着数据点发生器产生的簇在不同方向上有着不同的方差，不像之前那样点都集中在中心附近的圆形区域内，而是会形成一个以这个点为中心的椭圆形区域。下面，我们将在这些数据上以in-memory的形式运行k-means的实现。

图9-12显示了椭圆形分布（或称非对称分布）的数据点，以及由k-means产生的各个簇。很明显，k-means并不能挖掘出这些点的真实分布情况。

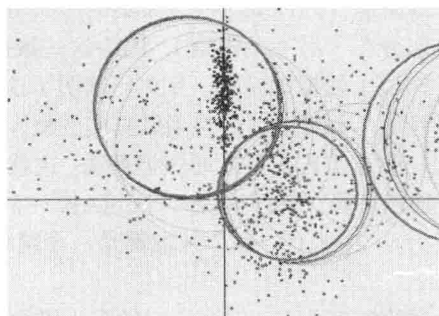


图9-12 在非对称正态分布的点集上运行k-means聚类。这些点分布在一个椭圆区域而不是一个圆形区域。k-means没能给出理想的结果

k-means的另一个问题是你需要估计*k*，即中心个数，而这个参数通常都会被估计得过高。找到最优的*k*值并不容易，除非你对数据有一个很明确的先验知识，但这种情况很少见。即使是做canopy生成，你也需要在距离测度方面进行调优，使得算法可以改善对*k*值的估计。

要是有一种更好的方式来确定簇个数，会怎么样呢？这就是基于模型的聚类方法的用武之地。

2. 对真实数据进行聚类所存在的问题

假设我们想基于一群人对电影的喜好来对他们进行聚类，以找出有共同爱好的人。我们可以统计电影的风格，以此来估计这群人中的簇个数。

我们找到的簇可能有：喜欢动作片的人、喜欢浪漫电影的人、喜欢喜剧的人等。但这并不理想，因为可能会有很多例外情况。例如，有一组人只喜欢犯罪片，而不喜欢其他动作片。他们就在动作片簇中形成了一个子簇。在簇的混合如此复杂的情况下，我们无法得到这个小簇的信息，因为它们通常都湮没在了大的簇中。唯一的解决办法是预先知道人们对电影喜好的内在层次结构。

如果我们预先知道这个情况，就可以通过层次聚类方法来得到更好的聚类效果。但是这类方法不能涵盖有重叠的情况。我们现在看到的所有聚类算法都不能同时处理既有重叠、又有层次结构的情况。那么我们应该怎么处理这类问题呢？

这是另一个可以通过基于模型的聚类方法解决的问题。

9.4.2　狄利克雷聚类

Mahout中有一个基于模型的聚类算法：狄利克雷聚类（Dirichlet clustering）。狄利克雷（Dirichlet）这个名字指代一类概率分布，它是德国数学家Johann Peter Gustav Lejeune Dirichlet定义的。狄利克雷聚类算法是一种基于狄利克雷分布的混合建模方法。

如果对狄利克雷分布没有一个深入了解的话，这个过程听起来很复杂，但它的思想其实很简单。假设你的数据点集中在一个类似圆形的区域内，它们在其中呈均匀分布，你也有一个描述该分布的模型。你可以读入数据，并计算该模型与数据相吻合程度，从而验证你的数据是否符合这种模型。

该方法可以在一定程度上自信地告诉你说这些点聚集的区域看起来像是圆形的分布，也可以说这个区域看起来不像是三角形分布（另一个模型），因为这些数据与三角形的吻合度太低。如果你找到一种拟合方式，也就知道了数据的结构。图9-13阐明了这一观点。（注意这里用到的圆和三角形仅仅是为了算法的可视化。不要把它们误当做真实的概率模型。）

狄利克雷聚类算法在Mahout中被实现为贝叶斯聚类算法。这就意味着，这个算法不仅仅是给出数据的一个解释，而是给出很多解释。就好比说，“区域A像一个圆，区域B像一个三角形，区域A和B合起来像一个多边形”等。实际上，每个区域都是一个统计分布，就像你在本章前面看到的正态分布一样。

我们接下来会见到各种各样的模型分布，但对它们的深入讲解超出了本书的范畴。我们还是先看看Mahout中的狄利克雷聚类算法是怎么工作的吧。

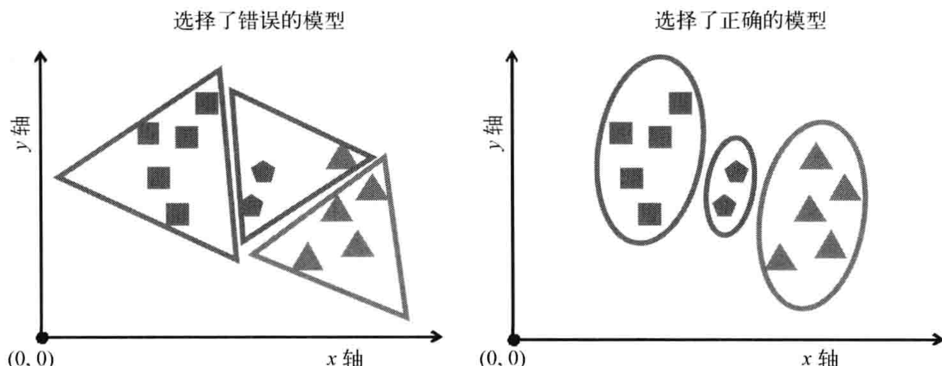


图9-13 狄利克雷聚类：这些模型尽最大可能去拟合给定的数据集。右边的模型要更好一些，它显示出了在这个模型下数据集中的簇个数

理解狄利克雷聚类算法

狄利克雷聚类的初始状态是一个数据点的集合以及一个`ModelDistribution`。可以把`ModelDistribution`看成是一个生成不同模型的类。你会创建一个空的模型，并尝试将点分配给它。然后，模型粗略地上下调整其参数，试着去拟合数据。当所有点上的拟合都完成之后，它会基于所有数据点以及点属于该模型的局部概率来精确地重新估计出模型参数。

在每个步骤末尾，你将得到一定数量的样本，其中包括概率、模型、点到模型的分配关系。这些样本可以被视为簇，它们提供了有关模型及其参数的信息，例如它们的形状和大小。另外，通过检查样本中被分配有数据点的模型个数，你可以知道数据支持多少个模型（簇）。还有，检查两个点分配给同一个模型的概率，你可以知道它们用同一模型表示时的相似度。这种软从属信息是使用模型聚类的一个副产品。狄利克雷聚类能够捕获到数据点属于多个模型的局部概率。

9.4.3 基于模型的聚类示例

基于狄利克雷过程聚类的in-memory版本在`DirichletClusterer`类中实现，而MapReduce作业的版本在`DirichletDriver`类中实现。我们将使用`generateSamples`函数（之前在9.1.2节用过）生成随机向量。

该狄利克雷聚类实现非常通用，可用于各种分布及数据类型。不过，Mahout中的`Model`实现使用了`VectorWritable`类型，所以在我们的聚类代码中把它作为默认类型。

我们用如下参数运行狄利克雷聚类：

- 输入`Vector`数据以`List<VectorWritable>`形式表示；
- 我们将数据拟合为`GaussianClusterDistribution`模型分布；
- 狄利克雷分布的alpha值为1.0；
- 初始模型数目（`numModels`）是10；
- `Thin`和burn间隔都是2。

数据点会分散到一个指定中心的周围，类似于正态分布，如下述代码清单所示。

代码清单9-8　对服从正态分布的数据进行狄利克雷聚类

```
List<VectorWritable> sampleData = new ArrayList<VectorWritable>();

generateSamples(sampleData, 400, 1, 1, 3);          ◁ 生成3个点集
generateSamples(sampleData, 300, 1, 0, 0.5);
generateSamples(sampleData, 300, 0, 2, 0.1);

DirichletClusterer dc =
  new DirichletClusterer(
    sampleData,
       new GaussianClusterDistribution(
         new VectorWritable(new DenseVector(2))),
       1.0, 10, 2, 2);                               运行狄利克雷聚类程序
List<Cluster[]> result = dc.cluster(20);             ◁
```

在代码清单9-8中，我们按照正态分布生成了一些样本点，并尝试使用正态（高斯）分布拟合数据。算法的参数决定了收敛的速度和质量。

这里，alpha是平滑因子。值越大模型的收敛速度越慢，所以聚类会有多次迭代，并且可能在这些模型上拟合过（模型会尝试在较小的范围内拟合数据分布，即使拟合了较大范围的模型正是我们想要的）。较小的alpha值会使得聚类过程中模型的合并更快，这样就会导致模型欠拟合（将来自两个不同分布的数据点拟合为一个模型）。

`Thin`和`burn`间隔用于降低聚类中内存的使用率。`Burn`参数指定要做多少次迭代才保存第一组模型。`Thin`参数指定每次保存这种模型配置之间要忽略多少次迭代。因为需要多次迭代才能收敛，初始状态没有进一步的利用价值。在初始化阶段，模型数量比较大，它们并不生成实际数据，所以我们可以忽略它们（通过`thin`和`burn`）以节省内存。

用`DisplayDirichlet`聚类算法得到的最终状态如图9-14所示。这个类放在Mahout的examples文件夹中，该文件夹中也有许多其他的模型分布及其聚类算法的例子。

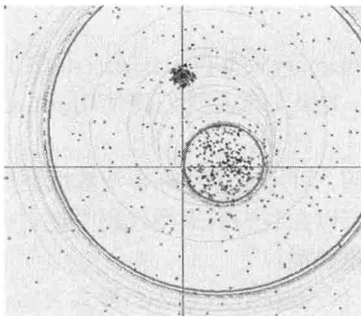


图9-14　使用`DisplayDirichlet`类对正态分布进行狄利克雷聚类，该类在Mahout的examples文件夹中

这里的聚类算法和9.2节的k-means聚类算法不同。狄利克雷聚类可以做一些k-means做不了的事情，比如可以精确定位我们生成的3个簇。其他算法只能找到重叠或者层次化的簇。

这只是冰山一角。为了展示基于模型的聚类的魔力，我们用一个比正态分布更加复杂的例子来说明。

1. 非对称正态分布

当数据点在不同维度的标准差不相等时，称正态分布为非对称的。这使得数据呈椭圆形分布。当你在这个9.4.1节的数据分布上做k-means聚类时，效果会很差。现在你可以在同样的数据集上用`GaussianClusterDistribution`完成狄利克雷聚类，这里在x和y轴上使用的参数不同。

你可以使用非对称征态分布模型执行狄利克雷聚类，这里二维平面内x和y方向上的标准差不同。聚类结果见图9-15。

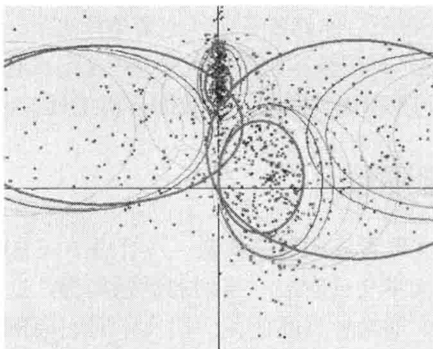


图9-15 非对称正态分布的狄利克雷聚类算法，所用`DisplayDirichlet`类的代码在Mahout的examples文件夹下。粗椭圆表示最终状态，细线表示之前的迭代过程

尽管生成的簇个数变多了，基于模型的聚类方法能够找到非对称的模型，并且相比起其他算法能够更好地在数据上拟合模型。使用较好的alpha值也许可以让结果得到改善。

下面，我们要尝试MapReduce版的狄利克雷聚类。

2. MapReduce版的狄利克雷聚类

就像Mahout中的其他实现一样，狄利克雷聚类专注于处理海量数据集。狄利克雷聚类的MapReduce版本是在`DirichletDriver`类中实现的。狄利克雷作业可以通过命令行方式在Reuters数据集上执行。

下面我们看看如何通过MapReduce作业完成狄利克雷聚类：

- Reuters数据集为`Vector`形式；
- 默认的模型分布类（`-md`）为`GaussianClusterDistribution`;
- 在这个作业中，创建的所有向量的`Vector`类的默认类型是`SequentialAccessSparse-Vector`;
- 该分布的`alpha0`值应该被设为`-a0 1.0`;
- 初始簇数量为`-k 60`;
- 算法的迭代次数为`-x 10`。

在这个数据集上通过Mahout启动器中的`dirichlet`程序运行该算法的过程如下：

```
$ bin/mahout dirichlet \
-i reuters-vectors/tfidf-vectors \
-o reuters-dirichlet-clusters -k 60 -x 10 -a0 1.0 \
-md org.apache.mahout.clustering.dirichlet.models.GaussianClusterDistribution \
-mp org.apache.mahout.math.SequentialAccessSparseVector
```

狄利克雷聚类的每轮迭代之后，作业把状态写到输出文件夹下的子文件夹中，命名形式为state-*。你可以用SeqenceFile读取程序读取它们，并且找到每个模型的中心点和标准差。由此，你可以在聚类的最后阶段将向量分配到各个簇。

当已知数据分布模型时，狄利克雷聚类是一个获得高质量簇的有力方式。在Mahout中，这个算法是一个可定制的框架，很容易创建并测试不同的模型。随着模型变得更加复杂，处理海量数据的进程会减缓，所以，在这种情况下，你就要考虑其他聚类算法了。但是通过观察狄利克雷聚类算法的输出，你可以决定我们应该选择模糊还是严格、重叠还是层次化的簇，距离测度选曼哈顿还是余弦，收敛的阈值是多少。狄利克雷聚类既是一个数据分析工具，也是一个有力的聚类工具。

9.5　用 LDA 进行话题建模

到目前为止，我们一直把文件看成词项的集合，并给每个词项赋予一定的权重。在实际生活中，我们把新闻文章或者其他文本文件看成一系列的话题集合。这些话题本质上是非常模糊的。少数情况下，甚至是有歧义的。通常，当我们读一段文字时，会把它和一系列话题联系在一起。如果有人问："这篇新闻讲的是什么？"我们很自然地会说"这讲的是美国的反恐战争"之类的话，而不是简单地把我们在文中看到的词列出来。

考虑一个话题，如狗。关于这个话题有很多文字描述，分别是在说不同的事情。在这些文档中最常出现的词也许是dog（狗）、woof（低吠声）、puppy（小狗）、bark（狗吠）、bow（弓腰）、chase（追赶）、loyal（忠诚）以及friend（朋友）等。有一些词，如bow（弓腰）和bark（狗吠）是有歧义的，因为它们也会出现在其他话题中，如弓（bow）和箭（arrow），或者一棵树的树皮（bark）等。但是，我们可以说上述这些出现在"狗"这个话题的词中有些词出现的概率比其他词要大。类似地，一个有关"猫"的话题中，诸如cat（猫）、kitten（小猫咪）、meow（喵）、purr（猫的呼噜声）和furball（毛绒球）等词就会频繁出现。图9-16给出了一些看到一幅有关狗或猫的图画之后人们心中会想起的词。

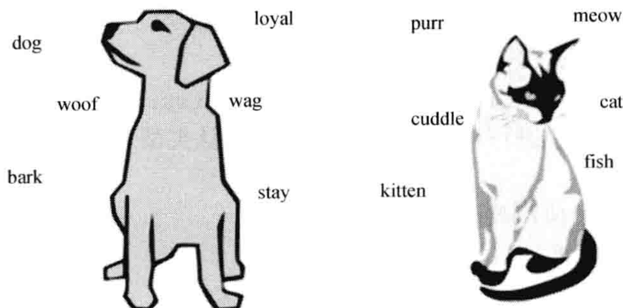


图9-16　猫狗话题以及其中频繁出现的词

如果要我们在特定的文档集中找出这些话题的话，直觉上肯定会用聚类思想。我们要调整聚类算法的代码，让它用在词语向量而不是之前用的文档向量上。这里的词语向量是指每个词对应一个向量，其特征是语料库中与这个词共现的其他词的ID，权重则是它们一起出现的文档数目。

一旦我们有了这个向量，就可以运行聚类算法了，找出词语的簇，并将其称为话题。尽管这看起来很简单，但生成词语向量的工作量相当大。我们可以把一起出现的词语归类为一个话题，然后计算词语在每个话题中出现的概率。

潜在狄利克雷分析（LDA）能做的不仅仅是聚类。如果两个词的意思或者形式相同，但是它们并不一起出现，聚类也不能基于其他的实例把它们关联起来。这就正是LDA大显身手的地方。LDA能够根据词语出现的方式来判断，指出哪些有相同的意思，或者被用在相同的上下文中。这类词语集合可被视为概念或者话题。

现在，我们扩展一下这个问题。假设我们有一系列观测结果（文档以及文档中的词）。我们能根据特征（话题）找出隐藏的分组吗？LDA可以非常高效地对特征进行分组或找到话题。

9.5.1 理解LDA

LDA是一个和狄利克雷聚类类似的生成式模型。首先，从一个已知的模型开始，然后通过调整参数，在数据上拟合模型。LDA假设整个语料库中有k个话题，并且每个文档都涉及这k个话题。因此，文档就可以看成不同概率的话题的混合体。

注意 机器学习算法分成两类——生成式（generative）和判别式（discriminative）。像k-means和层次聚类这类算法，就是基于距离测度，把数据划分成k个组，这种方法通常叫做判别式方法。判别式方法的一个例子是SVM分类，你将在本书第三部分学习。在狄利克雷聚类中，模型会对数据进行拟合，仅仅通过这些模型的参数，你就能生成拟合模型用的数据。因此，这个叫做生成式模型。

LDA比标准聚类方法更加强大的地方在于它既能把单词聚类成话题，也能把文档聚成多个话题的混合体。假设有一篇关于奥运会的文章，包括“金牌”“奖牌”“跑步”“冲刺”这些词；另一篇文章描述了亚运会的百米冲刺，包括“冠军”“金牌”“冲刺”这些词。LDA可以推断出一个模型，其中第一篇文章与两个话题都有关，其中一个讨论体育运动，包括“冠军”“金牌”“奖牌”等关键词；另外一个讨论百米冲刺，其中包括像“跑”和“冲刺”这类关键词。LDA能得出每个话题产生不同文档的可能性。这些话题本身就是这些词的分布，所以“跑”这个词出现在“体育运动”这个话题中的概率比出现在“百米冲刺”这个话题中的概率要低一些。

LDA算法和狄利克雷聚类工作方式类似。它首先创建一个空的主题模型，在mapper阶段并行读取文档，然后在文档中为每个单词计算每个主题的概率。然后，概率就会被发送给reducer，reducer会将这些概率相加并归一化。上述过程一直重复直到模型能够更好地表达这些文档——当概率（取对数）的和不再变化为止。变化程度是通过收敛阈值参数设定的，和k-means聚类的阈

值一样。LDA评估模型和数据的拟合程度，而不是测试中心点的相对变化值。如果似然值的变化小于这个阈值，则迭代停止。

9.5.2　对比TF-IDF与LDA

对文档进行聚类时，我们用TF-IDF单词权重来找出一个文档中重要的单词。TF-IDF有一个缺陷，它不能识别共同出现的单词或者它们之间的关系，如Coca和Cola。并且，TF-IDF不能通过单词的出现和分布找出这些单词之间的微妙关系。LDA可以以单词频率为输入，找出它们之间的关系，所以算法的输入应该是词频向量，而不是TF-IDF向量。

9.5.3　LDA参数调优

在运行Mahout中的LDA实现之前，你需要理解LDA中影响运行时间和质量的两个参数。

第一个是话题数目。就像k-means中的*k*个中心，你需要基于现有数据确定话题数目。如果话题数较少，就会得到更宽泛的话题，如科学、运动、政治，这些宽泛的话题会吞并其子话题中的单词。如果话题数目较多，每个话题就会更为专注或者小众，如量子力学和反射定律。

话题数目较多也意味着算法需要更长时间评估所有话题的单词分布。这会严重影响性能。根据经验，最好是具体问题具体分析。Mahout的LDA是MapReduce作业的形式实现的，所以它能运行在一个大的Hadoop集群上。你可以通过添加工作节点达到加速的效果。

第二个参数是单词的个数，也就是向量的维数。这决定了LDA的mapper中所用矩阵的大小。Mapper创建一个矩阵，其大小为话题个数乘以文档长度（文档中的单词或特征的数目）。

如果你需要加速LDA，除了可以降低话题数目外，还可以让特征数目最小化，但是如果你需要找到所有话题中所有单词的概率分布，你还是别动这个参数了。如果你只对语料库中的某些关键字感兴趣，你可以在创建向量的过程中剔除高频词汇。

你可以在基于词典的向量模型中降低maximum-document-frequency比例参数（`--maxDFPercent`），若取值为70，则会把在70%文档中都会出现的单词剔掉。

9.5.4　案例学习：寻找新闻文档中的话题

我们将在Reuters数据集上运行Mahout中的LDA。首先，我们运行词典向量化程序，创建TF向量，然后把它们用作LDADriver的输入。删除高频词汇以加速计算过程。在本例中，我们要从Reuter向量中提取10个话题。

LDADriver入口函数需要以下参数：

- 包含`Vectors`的输入目录；
- 每轮迭代之后写入LDA状态的输出目录；
- 模型的话题数目，`-k 10`；
- 语料库中的特征数量，`-v`；
- 话题平滑参数，`-a`（默认值为50/话题数）；

- 最大迭代次数限制，`-x 20`。

语料库中的特征数（`-v`）可以通过统计vectorizer文件夹下词典文件中的条目数得到。我们可以用`SequenceFileDumper`找到词典条目数，前面第8章有介绍。我们可以在命令行中运行LDA算法：

```
$ bin/mahout lda \
-i reuters-vectors/tf-vectors \
-o reuters-lda-sparse \
-k 10 -v 7000 -x 20 -ow
```

LDA会迭代20次，或者在收敛时就停止下来。每次迭代之后模型的状态就会被写入输出目录中以state-开头的文件夹内。

Mahout的`mahout-utils`模块有一个读取LDA输出的工具，用于从输出状态目录中读取话题和单词概率。`LDAPrintTopics`是该工具的主入口函数。你可以查看任何一轮迭代的输出，以下显示每个话题的前5个单词：

```
$ bin/mahout org.apache.mahout.clustering.lda.LDAPrintTopics \
-i reuters-lda-sparse/state-20/ \
-d reuters-vectors/dictionary.file-* \
-dt sequencefile -w 5
```

表9-1列出了这个例子的输出。注意10个话题只列出了5个。

表9-1　在Reuters新闻数据上进行LDA话题建模，各话题中的前5个单词

话题0	话题1	话题2	话题3	话题4
wheat	south	loans	trading	said
7-apr-1987	said	president	exchange	inc
agriculture	oil	bank	market	its
export	production	chairman	dollar	corp
tonnes	energy	debt	he	company

LDA能够从Reuters数据集中提取出多种话题集合。但是，仍然会出现一些没有太大实际意义的词，如7-apr-1987、said、he等。LDA将它们与集合中的其他词语同等对待。一般需要更多的迭代次数才能找到更好的话题模型。

想去掉这些不需要的词并不容易，因为他们出现的频率很高。相比起关键词来说，它们属于任何一个话题的概率都要更高。如果我们检查一下包含这些话题的文档，会发现这是显而易见的。但即使在词典向量中去掉高频单词之后，像said、he之类的词也是不会消失的。LDA能做得更好吗？

上面的例子中我们还有一个参数未修改过，即话题平滑参数（`-a`）。因为文本数据有许多噪声，这会导致LDA的估计出现错误。LDA可以通过增大平滑值来避开这个问题，这将增加那些低频关键词的权重。当然，这么做也会降低高频词的影响。这会导致LDA需要更多次的迭代才能产生有意义的话题模型。

默认情况下，LDA设置平滑参数为`50/numTopics`。在我们的例子中，它是`5`。现在我们把

这个平滑参数设为20，然后重新运行LDA。迭代结束之后，用LDAPrintTopics类来看输出。

输出列于表9-2中。高频词汇的影响仍然存在，但是话题内容变得更有条理了。

表9-2　增加平滑参数后，LDA话题建模所生成话题中的前5个单词

话题0	话题1	话题2	话题3	话题4
production	year	said	stock	vs
tonnes	growth	banks	corp	mln
price	foreign	have	securities	cts
oil	last	analysts	inc	net
department	billion	market	reuter	loss

9.5.5　话题模型的应用

话题模型的输出文件格式是“键–值”（IntPairWritable, DoubleWritable）。键（key）是一对整数，第一个是主题ID，第二个是特征ID；值（value）是单词在模型中出现的似然值。

这些话题模型可以用来解决很多实际问题。我们可以将他们用作簇中心，并使用距离测度将文档关联到最近的中心，也可以为它们分配标签，并将它们用作分类模型，这同样需要使用某种距离测度。

话题集合可以以标签云（tag cloud）的形式进行可视化，类似Digg和Delicious提供的功能。话题建模同样可以用于将话题按时间轴进行可视化。我们可以以月或者年为周期在新闻文章上建立话题模型。这样就能展示出话题随时间的发展趋势。

注意　一个有趣的实验是在时间轴上对科学研究进行话题建模。在19世纪90年代年左右的科学杂志中最经常出现的词是和蒸汽机相关的，20世纪40年代是与原子研究相关的，20世纪90年代则是与聚合物和半导体器件相关的。David M. Blei在如下链接中给出了这个实验的解释：http://www.cs.princeton.edu/~blei/topicmodeling.html.

话题模型中的词可以用来提高搜索的覆盖率。例如，一个人搜索可口可乐，可口可乐和百事可乐会同时出现在查询结果中。

LDA算法可以从语料库中发现有趣的簇，以及语料库中词语之间的关联。人们仍然在探索如何充分利用这些信息。Mahout中的LDA可以帮助我们在大量的服务器上分析数以百万计的文件。因为它的运行速度非常快，做起实验来非常方便。我们将在第12章的案例学习中探究LDA，展示如何用它来改进相关文件发掘的框架。

9.6　小结

在这一章中你了解了Mahout中的聚类算法。我们按聚类策略将各种聚类算法归纳到表9-3中。

表9-3 Mahout中的各种聚类算法、入口类及属性

算 法	in-memory实现	MapReduce实现	固定簇数	部分从属
k-means	KMeansClusterer	KMeansDriver	Y	N
canopy	CanopyClusterer	CanopyDriver	N	N
模糊k-means	FuzzyKMeansClusterer	FuzzyKMeansDriver	Y	Y
狄利克雷	DirichletClusterer	DirichletDriver	N	Y
LDA	N/A	LDADriver	Y	Y

Mahout实现的k-means算法对大小数据集都适用。估计出好的簇中心会使得聚类更快速，所以我们探索了改善中心估计的方法。canopy聚类算法可以快速、近似地对数据进行聚类，并得到近似的簇中心。使用这些中心作为初始值，k-means迭代过程会更快地收敛。我们也了解了k-means的各种参数，并用k-means创建了一个新闻网站聚类的模块。通过尝试Mahout中的各种距离测度类，我们可以优化新闻聚类模块，以得到更好的文本聚类质量。

模糊k-means聚类给出了文档对不同簇的部分从属信息，并且要比k-means具有更好的收敛性。我们使用模糊k-means作为聚类模块，来获取这种软分配信息。由于k-means和模糊k-means都需要预先设定k值，我们也探索了其他方法，并发现基于模型的聚类算法可以作为一个很好的替代品。

狄利克雷聚类是Mahout中基于模型的聚类算法，它并不仅仅是将点分配给各个簇，还可以解释模型对数据的拟合程度以及簇中点的分布情况。该算法可以找出很复杂的数据集中的簇，而前面提到的方法则不行。狄利克雷聚类被证明是描述此类数据的一个有力工具。

最后，我们介绍了LDA。LDA是聚类领域的一个新进展，可以将数据建模为混合话题。这些话题不仅仅是文档簇，也是词的概率分布。LDA可以在将单词集合聚类为话题的同时，将文档与多个话题关联起来。LDA带来了一些新的发展方向，它允许我们仅通过分析文本语料库就能得到单词之间的联系。

实际应用中，究竟哪种方法最适合你的数据，还要通过实验来选择。Mahout的聚类包中有一些有力的工具，它们建立在Hadoop的基础上，良好的可扩展性使得你可以通过简单添加机器来对任意规模的数据进行聚类。

下面的章节将更注重聚类算法速度和质量的调优。沿着这个思路，我们将优化新闻聚类代码，并最终展示一个实用的相关文章功能。我们也会在案例学习中探索一些有趣的问题，它们都可以用Mahout中的聚类算法来解决。

下面，我们将在第10章中介绍Mahout中一些不太常见的工具和技术，它们可以帮助你理解并改善聚类的质量。

第 10 章 评估并改善聚类质量

本章内容

- 检查聚类输出
- 评估聚类质量
- 改善聚类质量

前一章中我们学习了很多聚类算法：k-means、canopy、模糊k-means、狄利克雷聚类以及LDA等。它们在特定数据集上表现效果很好，而有时在其他数据集上却表现较差。完成任一聚类之后都会有一个很自然的问题：算法在这个数据集上的表现究竟如何？

分析聚类的输出是一项重要工作。可以通过简单的命令行工具，或可视化的GUI工具来完成。将簇可视化并确定问题所在之后，这些结果可以正式作为质量评价的指标，即以数值的形式来表明簇的好坏。在本章中，我们将介绍几种检查、评估和改善聚类算法的方法。

对聚类算法调优涉及建立自定义的距离测度标准以及选择合适的算法。评价指标反映出距离测度标准对聚类质量的影响。首先，你需要知道簇看起来的样子以及簇中心能够表达的特征。你还需要知道数据点在不同簇间的分布。你肯定不希望*k*-1个簇都只有一个数据点，而第*k*个簇则囊括了剩下所有的数据点。

知道簇的样子之后，你就能集中精力去改进聚类算法。为此，你需要知道如何调整输入向量的质量、距离测度标准以及算法的各种参数。

下面开始我们的旅程，去探索一些能帮助检查聚类输出的工具和技术。

10.1 检查聚类输出

Mahout中检查聚类输出的主要工具是`ClusterDumper`。`ClusterDumper`就在mahout-utils模块的org.apache.mahout.utils.clustering包中（也可通过bin/mahout脚本的`clusterdump`命令调用）。用`ClusterDumper`读取Mahout中聚类算法的输出很方便。第9章中介绍k-means时你就已经见过它了，本章中我们将用它来分析聚类质量。

`ClusterDumper`的输入是簇集合，此外，将数据转为向量时生成的词典也是一个可选的输入。`ClusterDumper`完整的选项列于表10-1中。

表10-1 Mahout中**ClusterDumper**工具的选项及默认值

选 项	标 志	描 述	默认值
SequenceFile目录（String）	-s	包含簇SequenceFile的目录	N/A
输出（String）	-o	输出文件；若不指定，则输出到终端	N/A
数据点目录（String）	-p	聚类结束之后，Mahout的聚类算法会产生两种输出。一种是成对的<簇id, 簇中心>集合，另一种是成对的<数据点id, 簇id>集合。后者在聚类完成时生成，通常存放在输出目录的points文件夹下。当此参数设为points文件夹时，会输出一个簇中的所有数据点	N/A
词典（String）	-d	词典文件路径，词典文件包含从整数ID到单词的反向映射	N/A
词典类型（String）	-dt	词典文件的类型。若为text（文本），则整数ID和单词应该用制表符分隔。若格式为SequenceFile，则应该有一个Integer类型的键和String类型的值	text
单词个数（int）	-n	需要打印的单词个数	10

回想一下，任何簇都有一个中心点，它是该簇中的所有点的平均值。除了基于模型的聚类算法如狄利克雷算法之外，这对所有算法都成立。在狄利克雷聚类过程中，中心并不是点的平均值，而是一组特定数据点的锚点。

中心点可以作为对簇的概括总结，方便我们快速了解各个簇的性质。中心点中权重最高的那些特征（即值最大的那些维度）最能反映簇的性质。对于文本文档来说，特征即是单词，也就是说中心点权重最高的那些单词反映出了该簇中文档要表达的含义。

例如，讨论近期美国总统大选的文档簇中，Obama、McCain和election这些特征将会有较高的权重。这些权重最高的特征之间在语义上也是有关联的。另一方面，如果权重最大的特征并不相关，则意味着簇中记录了一些不相关的东西。导致这一现象的原因可能是多方面的，本章会对其中的大部分原因进行讨论。

我们来看一个例子，在两个k-means聚类输出上执行ClusterDumper的结果：其中一个使用了欧氏距离测度，而另一个使用了余弦距离测度：

```
$ bin/mahout clusterdump \
-s kmeans-output/clusters-19/ \
-d reuters-vectors/dictionary.file-0 \
-dt sequencefile -n 10
```

ClusterDumper为每个簇输出了一系列信息，包括中心向量和簇中权重最高的词汇。向量的各个维度被转化为词典中的单词并输出到屏幕上。将这些输出写入到文本文件中可能更容易查阅，可以使用-o选项来实现这一功能：

```
$ bin/mahout clusterdump \
-s kmeans-output/clusters-19/ \
-o output.txt \
-d reuters-vectors/dictionary.file-0 \
-dt sequencefile -n 10
```

可以使用任何文本编辑器打开这里的output.txt文件。包含各簇的中心向量的行通常比较长，因此在下面的输出中将其省略。下面的代码清单是一个典型的top-10词汇有序列表，取自Reuters数据集上k-means聚类的结果，其中使用了欧氏距离测度。

代码清单10-1　k-means聚类结果中的前10个词（使用欧氏距离测度）

```
Top Terms:
                    said                    =>   11.60126582278481
                    bank                    =>   5.943037974683544
                    dollar                  =>    4.89873417721519
                    market                  =>   4.405063291139241
                    us                      =>  4.2594936708860756
                    banks                   =>  3.3164556962025316
                    pct                     =>   3.069620253164557
                    he                      =>   2.740506329113924
                    rates                   =>  2.7151898734177213
                    rate                    =>  2.7025316455696204
```

上述簇中靠前的词汇显示它包含与银行有关的文档。一个类似的簇列于代码清单10-2中，它是使用余弦相似度得到的。

代码清单10-2　k-means聚类结果中的前10个词（使用余弦距离测度）

```
Top Terms:
                    bank                    =>  3.3784878432986294
                    stg                     =>   3.122450990639185
                    bills                   =>   2.440446103514419
                    money                   =>   2.324820905806048
                    market                  =>   2.223828649332401
                    england                 =>  1.6710182027854468
                    pct                     =>  1.5883359918481277
                    us                      =>   1.490838685054553
                    dealers                 =>  1.4752549691633745
                    billion                 =>  1.3586127823991738
```

它们很相似，但看起来后者要更好一点。第二个簇中*money*是一个重要词汇，而第一个簇的重要词汇中出现了*said*这个常见动词。

下一节将基于上述输出来对聚类质量进行评估。

10.2　分析聚类输出

影响聚类质量的不仅仅是输入，还包括众多的算法，每个算法都有需要调优的参数。如果聚类出现了错误，要对其进行调试是很困难的。因此，通过分析簇来了解发生了什么显得格外重要。

我们首先来分析代码清单10-1和代码清单10-2的输出，主要关注以下3个方面：

- 距离测度和特征选择；
- 簇内以及簇间距离；
- 混合以及重叠的簇。

10.2.1 距离测度与特征选择

对于文本来说，余弦距离测度更为合适，因为它根据高权重的公共单词来聚合文档。TF-IDF权重向量中，话题单词具有较高的权重，因此使用余弦距离测度聚合到一起的相似文档更倾向于拥有公共的话题单词。这使得簇中心向量中，话题单词相比停用词会具有更高的平均权重。

代码清单10-2中的重要单词都是话题相关的，而且没有像*said*这样的停用词。我们在代码清单10-1中则遇到了这类词，即包含了一些并不重要的词汇。尽管两个示例中都使用了TF-IDF权重向量，代码清单10-1的方法中，*said*这个常见动词的重要性较高。尽管*said*的权重（在TF-IDF向量中）较小，其在整个簇中得到的均值却会比一些话题单词还要大，因为欧氏距离测度给予所有的特征同样的重要性。

如你所见，选择好的距离测度有助于改善质量。

对特征进行加权

好的簇通常围绕一些较强的特征而形成，这些特征使得文档之间在概念上形成强烈的相似性。这意味着选择正确的特征与正确的距离测度有着同样的重要性；对于无结构的文本，需要提取好的特征以得到好的聚类质量。但TF-IDF加权有一些局限性。对于实际应用，我们需要在这一加权技术的基础上加以改进。

对于其他类型的数据，需要对权重向量进行精心设计以得到最优结果。好的聚类算法需要给予重要特征较高的权重，而给予不重要的特征较低的权重。

回忆第8章中对苹果向量化的例子。从数值上来讲，颜色值较大，重量值较小。如果我们发现重量是比颜色更好的特征，那么重量值就需要放大，而颜色值则需要缩小。例如，颜色值可以缩小到0~1之间，而重量值可以放大到0~100之间。这样一来，这些值的物理意义就不存在了。留下的仅仅是一个向量，其特征和权重会使得对苹果的聚类效果更佳。

10.2.2 簇间与簇内距离

给出所有中心点，可以计算出特定距离测度下所有中心点对之间的距离，并以矩阵形式表示出来。这个簇间距离矩阵很好地反映了聚类过程中所发生的事情，它展示了最终结果里簇之间的远近关系。

簇内距离是一个簇内部所有成员间的距离，而不是两个不同簇之间的距离。这一指标反映出距离测度标准汇聚元素的能力。

下面我们来看看这两类距离。

1. 簇间距离

簇间距离能够很好的反映聚类质量；好的聚类结果中不同簇中心点之间不大可能靠得很近，太近则意味着聚类过程产生了多个具有相似特征的组，并导致簇之间的区别不够显著。

如果找到一篇新闻，它在关于US、president和election的簇中，而不在讨论candidate、United States和McCain的簇中，这有意义吗？应该没有；我们可能不希望簇相互之间隔得太近。簇间距离与这方面的质量密切相关。图10-1显示了两种不同数据分布的簇间距离。

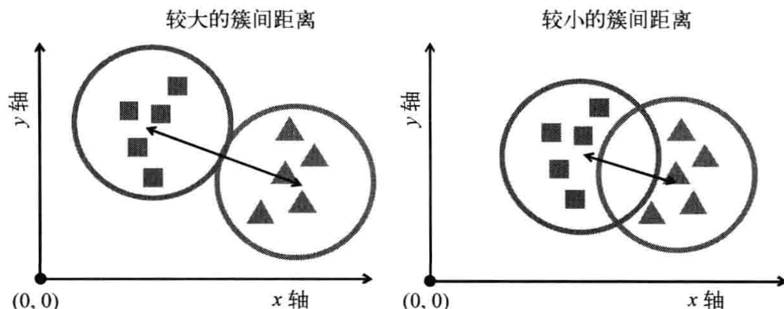


图10-1 簇间距离是反映数据划分好坏的一个标准。它依赖于特征加权技术以及所用的距离测度

我们知道欧氏距离值通常要比余弦距离值大。在距离测度不同的情况下比较簇间距离没有意义，除非他们都被归一化到同样的范围。归一化后的所有簇的平均簇间距离能够很好的反映聚类质量。

下面的代码清单给出了一种根据聚类输出计算簇间距离的简单方法。

代码清单10-3 计算簇间距离

```
public class InterClusterDistances {
  public static void main(String args[]) throws Exception {
    String inputFile
        = "reuters-kmeans-clusters/clusters-6/part-r-00000";      设定聚类输出路径

    Configuration conf = new Configuration();
    Path path = new Path(inputFile);

    System.out.println("Input Path: " + path);
    FileSystem fs = FileSystem.get(path.toUri(), conf);

    List<Cluster> clusters = new ArrayList<Cluster>();

    SequenceFile.Reader reader = new SequenceFile.Reader(fs, path, conf);
    Writable key = (Writable) reader.getKeyClass().newInstance();
    Writable value = (Writable) reader.getValueClass().newInstance();

    while (reader.next(key, value)) {                 读入簇对象
      Cluster cluster = (Cluster) value;
      clusters.add(cluster);
      value = (Writable) reader.getValueClass().newInstance();
    }

    DistanceMeasure measure = new CosineDistanceMeasure();
    double max = 0;
    double min = Double.MAX_VALUE;
    double sum = 0;
    int count = 0;
    for (int i = 0; i < clusters.size(); i++) {                 计算距离测度
      for (int j = i + 1; j < clusters.size(); j++) {
        double d = measure.distance(clusters.get(i).getCenter(),
            clusters.get(j).getCenter());
        min = Math.min(d, min);
        max = Math.max(d, max);
```

```
          sum += d;
          count++;
        }
      }

      System.out.println("Maximum Intercluster Distance: " + max);
      System.out.println("Minimum Intercluster Distance: " + min);
      System.out.println("Average Intercluster Distance(Scaled): "
          + (sum / count - min) / (max - min));
    }
  }
```

代码清单10-3计算了所有中心点对之间的距离，并计算了最小、最大以及归一化的簇间距离。对于使用CosineDistanceMeasure的Reuters聚类结果，它提供了如下输出：

```
Maximum Intercluster Distance: 0.9577057041381253
Minimum Intercluster Distance: 0.22988591806895065
Average Intercluster Distance(Scaled): 0.7702427093554679
```

在使用欧氏距离测度的Reuters聚类结果上运行同样的代码，会得到明显不同的输出：

```
Maximum Intercluster Distance: 96165.06236154583
Minimum Intercluster Distance: 2.7820472396645335
Average Intercluster Distance(Scaled): 0.09540179823852128
```

注意，使用余弦距离测度时，最小和最大簇间距离的大小比较接近。这意味着使用余弦距离测度时，簇的分布较为均匀。而使用欧氏距离时，最小簇间距离非常小，这意味着至少有两个簇几乎一样，或者说几乎完全重合。

归一化后的平均簇间距离清晰的显示出余弦距离测度要优于欧氏距离，因为它生成的簇分布更均匀。这也揭示了为什么基于余弦距离得到的簇中，重要词汇的质量要比基于欧氏距离得到的簇高。

2. 簇内距离

簇内距离（一个簇内的成员之间的距离）要比簇间距离小。图10-2展示了使用两种不同距离测度所得到的簇内距离。一个好的距离测度会使得相似对象间的距离较小，并产生更为紧凑的簇，因而也能更可靠的区分不同簇。



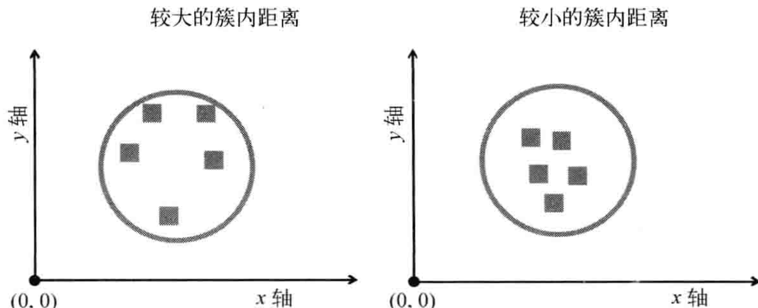


图10-2 簇内距离是反映数据点接近程度的指标。聚类质量取决于两个因素：距离测度给相隔较远的对象较大的惩罚值，给相隔较近的对象较小的惩罚值。二者的比值越大，簇越分散

10.2.3 簇的混合与重叠

在某些数据集中，很难将数据点划分为好的簇。这种情况下产生的簇具有较低的簇间距离，其大小已经接近簇内距离了。对这类数据集，更改距离测度或改进特征选择策略通常不会有什么效果。我们需要使用一些特殊的算法，例如模糊k-means（识别部分从属关系）或狄利克雷过程聚类（识别能够拟合数据的模型）。

回顾一下各种聚类算法是必要的，包括它们的优势以及适用的数据类型。下一节将探索一些能够改进簇间和簇内距离的进阶技术。

10.3 改善聚类质量

有两个因素对改善聚类质量至关重要：改进文档中特征的加权方式以及设计更合理的距离测度。好的加权策略可以突出对象中好的特征，而合适的距离测度则有助于将相似的特征聚集到一起。接下来的两个小节会告诉你如何设计自定义的特征选择类和距离测度类。

10.3.1 改进文档向量生成过程

好的文档向量要有合适的特征，即赋予重要特征更高的权重。在文本数据中，有两种方式可以用来改善文档向量的质量：去除噪声和使用合适的加权技术。

并非所有文本文档都是高质量的，可能会有一些比较怪异的语句，单词也可能由于格式或结构的原因而很难区分。互联网上产生的海量数据集就是如此。因特网上的大部分文本内容，如网页、博客、wiki、聊天记录或论坛，在传输过程中都混杂着大量标记、样式表和脚本，而不是纯粹的文本。通过OCR（Optical Character Recognition，光学字符识别）技术解析扫描文档而产生的文字，还有SMS信息，它们的质量都很差，例如字符识别错误（the识别为tne）或极端的俚语（用c u表示see you）。文本中可能丢失字符、空格或标点符号，或者包含随意的术语甚至意想不到的单词和语法错误。要在噪声如此严重的情况下完成聚类是很困难的。要得到比较好的质量，需要首先从数据中清理掉这些错误。

一些现成的文本分析工具可以很好地处理这些问题。Mahout提供了一个钩子，允许在向量化过程中注入任何文本过滤技术。这是通过一种叫做Lucene `Analyzer`的东西实现的。尽管当时我们没有做出太多解释，实际上你已经在第九章中接触过了Lucene `Analyzer`，在那里我们试图改进新闻聚类模块。

建立自定义的Lucene Analyzer包括如下步骤：

- 扩展`Analyzer`接口；
- 重载`tokenStream(String field, Reader r)`方法。

代码清单10-4展示了一个自定义的Lucene `Analyzer`，它使用`StandardTokenizer`将一篇文档词条化。`StandardTokenizer`是Lucene中实现的具有一定容错性的词条化工具。

代码清单10-4　一个封装了StandardTokenizer的自定义Lucene Analyzer

```
public class MyAnalyzer extends Analyzer {          <—— 扩展Analyzer

  @Override
  public TokenStream tokenStream(
      String fieldName, Reader reader) {            <—┐ 从Reader产生
    TokenStream result = new StandardTokenizer(        │ TokenStream

        Version.LUCENE_CURRENT, reader);
    return result;
  }
}
```

该自定义分析器将Reader输入的文本符号化为一个词条流。一个词条就是一个单词，所以你可以将一个文本文档视为一个词条流。在代码清单10-4中我们使用StandardTokenizer来将Reader中的内容词条化。

下一步我们来尝试改进这个Tokenizer。我们不再简单的将单词词条化，而是引入了一个过滤器来将词条都转为小写。更新后的类示于下面的代码清单中。

代码清单10-5　带有小写过滤器的MyAnalyzer

```
public class MyAnalyzer extends Analyzer {

  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
    TokenStream result = new StandardTokenizer(
        Version.LUCENE_CURRENT, reader);
   result = new LowerCaseFilter(result);          <—— 应用小写过滤器
   return result;
  }
}
```

TokenFilter对底层的词条流应用了某种变换，这些过滤器可以级联起来。除了LowerCaseFilter，Lucene还提供了StopFilter、LengthFilter和PorterStemFilter。StopFilter跳过底层词条流中的停用词，LengthFilter过滤掉长度不在指定范围内的词条，PorterStemFilter则取底层单词的词干。取词干是一个将单词还原到它的基本形式的过程。例如单词kicked和kicking都会被还原为kick。

注意　PorterStemFilter仅对英文文本有效，但Lucene中也有一些可用于其他语言的词干过滤器。

使用这些过滤器可以创建一个自定义的Lucene Analyzer，如代码清单10-6所示。

代码清单10-6　使用多个过滤器的自定义Lucene Analyzer

```
public class MyAnalyzer extends Analyzer {

  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
```

```
    TokenStream result = new StandardTokenizer(
       Version.LUCENE_CURRENT, reader);
    result = new LowerCaseFilter(result);
    result = new LengthFilter(result,3, 50);
    result = new StopFilter(result,
        StandardAnalyzer.STOP_WORDS_SET);
    result = new PorterStemFilter(result);
    return result;
  }
}
```

应用小写过滤器

根据长度删除单词

去除停用词

取单词词干

使用这个Lucene `Analyzer`，我们可以生成TF-IDF向量，并对Reuters集合中的数据进行聚类。这里我们借鉴了代码清单9-4中的代码，并引入了这里的`Analyzer`，如代码清单10-7所示。

代码清单10-7　修改NewsKMeansClustering.java，改用`MyAnalyzer`

```
public class NewsKMeansClustering {
  public static void main(String args[]) throws Exception {
    int minSupport = 5;
    int minDf = 5;
    int maxDFPercent = 99;
    int maxNGramSize = 1;
    int minLLRValue = 50;
    int reduceTasks = 1;
    int chunkSize = 200;
    int norm = -1;
    boolean sequentialAccessOutput = true;

    String inputDir = "reuters-seqfiles";
    File inputDirFile = new File(inputDir);
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);
    String outputDir = "newsClusters";
    HadoopUtil.delete(conf, new Path(outputDir));
    Path tokenizedPath = new Path(outputDir,
        DocumentProcessor.TOKENIZED_DOCUMENT_OUTPUT_FOLDER);
    MyAnalyzer analyzer = new MyAnalyzer();
    DocumentProcessor.tokenizeDocuments(
       new Path(inputDir), analyzer.getClass()
      .asSubclass(Analyzer.class), tokenizedPath, conf);

    DictionaryVectorizer.createTermFrequencyVectors(
        tokenizedPath,
        new Path(outputDir), conf, minSupport, maxNGramSize,
        minLLRValue, 2, true, reduceTasks,
        chunkSize, sequentialAccessOutput, false);
    TFIDFConverter.processTfIdf(
        new Path(outputDir,
            DictionaryVectorizer.DOCUMENT_VECTOR_OUTPUT_FOLDER),
        new Path(outputDir) , conf, chunkSize,
        minDf, maxDFPercent, norm, true, sequentialAccessOutput,
        false, reduceTasks);

    Path vectorsFolder = new Path(outputDir,
```

初始化`MyAnalyzer`

使用`MyAnalyzer`词条化

生成TF向量

生成IDF向量

```
                          "/vectors");
    Path centroids = new Path(outputDir,  "/centroids");
    Path clusterOutput = new Path(outputDir,  "/clusters");

    RandomSeedGenerator.buildRandom(conf, vectorsFolder, centroids,
      20, new CosineDistanceMeasure());

    KmeansDriver.runJob(conf, vectorsFolder, centroids,        <-- 使用随机中心的
      clusterOutput, new CosineDistanceMeasure(), 0.01,            k-means
      20, true, false);

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path(clusterOutput, Cluster.CLUSTERED_POINTS_DIR
            + "/points/part-m-00000"), conf);

  }
}
```

这里的代码与你在第8章和第9章中见到的很相似。区别是我们不再使用StandardAnalyzer，而是使用MyAnalyzer，然后在生成的向量上执行k-means聚类。在Reuters数据上运行完代码清单10-7的聚类代码之后，银行簇中的前10个词汇如下所示：

```
Top Terms:
                billion         =>  4.766493374867332
                bank            => 2.2411748607854296
                stg             => 1.8598368697035639
                money           => 1.7500053112049054
                mln             => 1.7042239766168474
                bill            => 1.6742077991552187
                dlr             => 1.5460346601253139
                from            => 1.5302046415514483
                pct             => 1.5265302869149873
                surplus         => 1.3873321058744208
```

注意，单词都用PorterStemFilter变成了词干，StopwordsFilter则保证了几乎不存在停用词。这体现出选择合适的特征集合对于改进聚类质量的意义。

10.3.2 编写自定义距离测度

如果向量的质量已经很好了，那就应该考虑选择合适的距离测度来改进聚类质量。我们已经知道余弦距离对于文本文档聚类来说较为理想。为了展示自定义距离测度的威力，我们设计了一个增强的余弦距离测度：它使大的距离更大，小的距离更小。与编写Analyzer类似，编写一个自定义DistanceMeasure包括如下步骤：

- 实现DistanceMeasure接口；
- 在distance(Vector v1, Vector v2)方法中编写距离测度。

我们更改后的余弦距离测度如代码清单10-8所示。

代码清单10-8 一种改进的余弦距离测度

```
public class MyDistanceMeasure                         实现DistanceMeasure接口
    implements DistanceMeasure {                    <--
```

```
  @Override
  public double distance(Vector v1, Vector v2) {                  <-- 重载distance方法
    if (v1.size() != v2.size()) {
      throw new CardinalityException(v1.size(), v2.size());
    }
    double lengthSquaredv1 = v1.getLengthSquared();
    double lengthSquaredv2 = v2.getLengthSquared();

    double dotProduct = v2.dot(v1);

    double denominator = Math.sqrt(lengthSquaredv1)
                       * Math.sqrt(lengthSquaredv2);

    if (denominator < dotProduct) {
      denominator = dotProduct;
    }
    double distance = 1.0 - dotProduct / denominator;      <-- 计算距离
    if (distance < 0.5) {
        return (1 - distance) * (distance * distance)
            + distance * Math.sqrt(distance);
    } else return Math.sqrt(distance);                     <-- 返回距离测度
  }

  @Override
  public double distance(double centroidLengthSquare,
                         Vector centroid,
                         Vector v) {
    return distance(centroid, v);
  }

  @Override
  public void createParameters(String prefix, JobConf jobConf) {}

  @Override
  public Collection<Parameter<?>> getParameters() {
    return Collections.emptyList();
  }

  @Override
  public void configure(JobConf arg0) {}
}
```

在一个MapReduce作业中，`configure()`、`getParameters()`和`createParameters()`等方法用来在各个作业节点中为`DistanceMeasure`传递参数。这里我们重点关注`distance`函数。它首先计算标准余弦距离测度，然后又通过取平方使得近的距离变得更近（若值在0到0.5之间），而使远的距离更远。

在此距离测度下，一些有趣的小众话题浮出水面，这是之前不曾出现过的：

```
排名靠前的词项:
                futures          =>  3.2517230513480757
                trading          =>    2.73124880187049
                exchange         =>  2.2752173132458906
                market           =>  2.0752484701513274
                said             =>  1.9316076421017707
                stock            =>  1.8062251904561821
```

```
new                  =>  1.5571645992558176
traders              =>   1.531255338250137
index                =>  1.3630116680911748
options              =>   1.183384693735014
```

注意，这个结果是从StandardAnalyzer词条化的向量中生成的。你可以尝试用自定义Lucene Analyzer和自定义的DistanceMeasure来得到理想的聚类结果。

10.4 小结

在本章中，我们探讨了如何改进聚类。算法、距离测度以及数据类型都会影响聚类的输出。ClusterDumper工具可以检查任何聚类算法的输出：各个簇的主要特征以及中心向量。这可以帮助你理解聚类过程。

对于海量数据集，手动的检查所有簇是不可行的。我们可以通过簇间和簇内距离来快速得到聚类质量的评分。如果很多簇都靠得太近，就需要考虑传统k-means以外的方法，看看部分从属关系或混合分布是否适用。

编写一个自定义的Lucene Analyzer和一个自定义的相似性度量，有助于在标准实现的基础上进一步改善聚类质量。我们发现通过尝试自定义度量标准来改善聚类质量是很有意义的。

现在，你已经有能力处理任何类型的数据，并调优你的聚类算法，使其更好地工作。很快你将遇到聚类中最大的困难：规模。幸运的是，Mahout可以将TB级的数据分散到庞大的Hadoop集群上进行处理。下面，我们会探索Mahout中的聚类算法如何发挥Hadoop的作用。在第11章中，你将学习如何在Hadoop集群上运行一个聚类作业，并使你的聚类算法应用于具体产品。

第 11 章

将聚类用于生产环境

本章内容

- 在Hadoop集群上运行聚类作业
- 对聚类作业进行性能调优
- 批聚类及在线聚类

前面我们已经看到，如何利用不同的Mahout聚类算法对路透社新闻数据集中的文档进行聚类。与此同时，我们也学到了数据的向量表示、距离测度方法和其他多种可以提高簇质量的方法。Mahout的一个优点是它的可扩展能力。路透社数据集几乎不具备挑战性，因此本章会给Mahout找一个更具挑战性的任务。我们将对世界上极为庞大的免费数据集——维基百科（Wikipedia）这部免费百科全书进行聚类。Mahout能够处理这种规模的数据，这是因为其算法以MapReduce作业的方式实现，而这些作业能够在成百上千台计算机构成的Hadoop集群[1]上运行。

遗憾的是，并非所有人都能访问这样一个集群。在本章中为示范起见，我们使用了从维基百科中提取的一个文档子集，并在一个小规模的集群上进行了实验，该实验能够说明通过增加机器来获得更快的速度。我们一开始在一个单机的Hadoop环境（也称伪分布式Hadoop，具体内容可以参见第6章）。我们还将考察Mahout的启动程序，它能够很容易地在本地或给定配置文件的条件时在任意Hadoop集群上启动聚类作业。然后，我们讨论如何对聚类作业进行性能调优。最后，我们讨论如何利用现有的Mahout聚类算法来设计一个能够以在线模式进行增量式聚类的系统。

11.1 Hadoop 下运行聚类算法的快速入门

首先让我们来看看Hadoop的架构。

Hadoop集群由一个叫做名字节点（NameNode，也称主节点）的服务器及其控制的不同数据节点（DataNode）组成。名字节点也用于同步Hadoop分布式文件系统（HDFS）。另一个称为JobTracker的服务器，负责管理所有的MapReduce任务以及集群中执行`Mapper`和`Reducer`的计算

① Ajay Anand在其博文“Scaling Hadoop to 4000 nodes at Yahoo!”中描述了2008年雅虎在4000个节点上运行Hadoop的情况，该博文的地址为http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop _to_4000_nodes_a.html。

机节点。在每一个节点当中，一个称为TaskTracker的进程负责管理着来自JobTracker的`Mapper`或`Reducer`执行请求。

Hadoop能够无缝地实现上述过程，并且不需要任何用户干预。在单节点集群中，名字节点、JobTracker、数据节点和TaskTracker等都运行在同一系统下，它们各自运行在独立的进程中并彼此交互。

本书写作之时，Mahout设计时计划使用的Hadoop版本是0.21（不过它应该与更新的版本兼容）。

11.1.1 在本地Hadoop集群上运行聚类算法

读者可以在网页http://wiki.apache.org/hadoop/QuickStart中找到一个建立本地伪分布式模式下的Hadoop集群的快速入门。你需要为本章的例子建立这种本地集群。

Hadoop二进制执行文件存放在Hadoop主目录的/bin目录下。要浏览HDFS的内容，执行如下命令：

```
bin/hadoop dfs -ls /
```

上述命令将会列出文件系统根目录下的所有文件。

当集群第一次启动时，你的主目录可能在HDFS中不存在，因此需要创建一个/user/<你的Unix用户名>的目录：

```
bin/hadoop dfs -mkdir /user/<你的Unix用户名>
```

也可以通过如下命令查看主目录：

```
bin/hadoop dfs -ls
```

要开始在你当前的伪分布式集群下对路透社数据进行聚类，需要像第8章一样准备包含路透社本地文本数据的`SequenceFile`文件，并将它复制到HDFS中：

```
bin/hadoop dfs -put <path-to>/reuters-seqfiles reuters-seqfiles
```

利用上述`SequenceFile`作为输入，就可以在集群上运行基于词典的向量化工具然后运行k-means聚类算法。

任意MapReduce作业都使用`hadoop jar`命令来执行。Mahout将所有的示例类文件及其依赖关系都打包放在examples/target/mahout-examples-0.4-SNAPSHOT.job下的单个JAR文件中。要在本地Hadoop集群下运行词典向量化工具来处理路透社`SequenceFile`输入文件，只需要简单地对Mahout作业文件运行如下hadoop `jar`命令：

```
bin/hadoop jar mahout-examples-0.5-job.jar \
  org.apache.mahout.vectorizer.SparseVectorsFromSequenceFiles \
  -ow -i reuters-seqfiles -o reuters-vectors
```

好，就是这样，聚类算法在本地Hadoop集群下运行。如果使用默认的Hadoop配置，并且如果系统至少是双核处理器，那么就有两个Mapper并行执行，其中每个Mapper运行在一个核上。这可以在JobTracker的面板（地址为：http://localhost:50030）上查看，具体的结果如图11-1所示。

lappy Hadoop Map/Reduce Administration

State: RUNNING
Started: Sun May 02 14:29:32 IST 2010
Version: 0.20.2-dev, r
Compiled: Mon Feb 8 03:33:04 IST 2010 by robinanil
Identifier: 201005021429

Cluster Summary (Heap Size is 26.19 MB/1.74 GB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
0	1	7	1	2	2	4.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
			mahout-examples-0.4-				

图11-1　一个典型的Hadoop MapReduce JobTracker的页面截图。可以通过访问本地Hadoop集群上的http://localhost:50030地址来访问该页

11.1.2　定制Hadoop配置

如果有两个核，那么Hadoop立马就可以将聚类的本地运行速度提高到两倍，关于本地运行在第8到第10章都有所介绍。如果计算机上有不止两个核，可以修改Hadoop配置文件将Mapper和Reducer任务的数目设置成一个更高的值，这样可以提高计算的整体运行速度。安装后的Hadoop中有mapred-site.xml配置文件，需要根据CPU核的数目，将其中的`mapred.map.tasks`和`mapred.reduce.tasks`配置项修改为合适的值。

每增加一个核，并行程度就增加，运行时间就会减少。一旦并行任务数达到单节点的峰值，那么再增加任务就会严重降低处理的性能。进一步扩展的唯一办法就是拥有配置相同的多个节点，即一个完整的分布式Hadoop集群。

提示　可以在Hadoop网站的如下地址http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html找到一个一步步建立分布式Hadoop集群的快速入门。在0.20.2及更高的Hadoop版本中，配置文件被分成conf目录下的三个文件：core-site.xml、hdfs-site.xml和mapred-site.xml。默认的配置值分别在core-default.xml、hdfs-default.xml及mapred-default.xml中。默认文件给定的参数可以被*.site.xml文件中的值覆盖。调整MapReduce参数通常涉及对mapred-site.xml文件的编辑。

在分布式Hadoop集群下运行聚类代码与在单节点下没有什么不同。bin/hadoop脚本能够在集群中启动各种作业，这些作业可以来自名字节点、从属节点或者任意可以访问名字节点的计算机。唯一的要求就是脚本在运行时要通过HADOOP_CONF_DIR环境变量访问正确的配置文件。

使用Mahout启动脚本在Hadoop集群下执行作业

Mahout也提供了一个和Hadoop启动脚本非常像的bin/mahout脚本来启动聚类作业。在前面章节中，该脚本被广泛用于以单进程作业的方式启动聚类过程。通过设置HADOOP_HOME和HADOOP_CONF_DIR环境变量，上述脚本可以用于在Hadoop集群上启动任意Mahout算法。该脚本将自动读取Hadoop集群的配置文件并在集群下启动Mahout作业。

一个典型的输出结果如下所示：

```
export HADOOP_HOME=~/hadoop/
export HADOOP_CONF_DIR=$HADOOP_HOME/conf
bin/mahout kmeans -h

running on hadoop, using HADOOP_HOME=/Users/username/hadoop and
     HADOOP_CONF_DIR=/Users/username/hadoop/conf
...
```

Mahout启动脚本通过使用正确的集群配置文件在内部调用了Hadoop的启动脚本，整个过程如图11-2所示。在Mahout中，启动脚本是一种在本地或者分布式Hadoop集群下启动算法的最简单的方式。

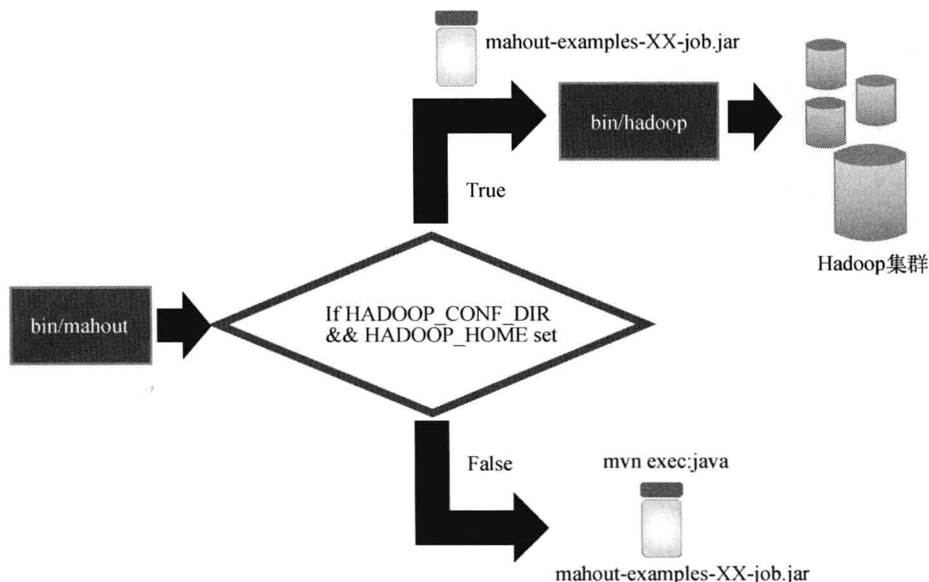


图11-2 利用Mahout脚本启动的两种作业执行方式的逻辑框图

到这里要向你表示祝贺，因为你已经在Hadoop下运行了一个聚类作业。如果感兴趣的话，你可以用多组参数进行实验来调节聚类的质量。接下来，我们将会看到如何通过调节配置来获得更高的聚类吞吐量和性能。

11.2 聚类性能调优

Mahout中的聚类算法被设计成并行执行。尽管不同算法之间有差别，但它们在某个方面是极其类似的，即在每个Mapper中它们都从SequenceFile文件并行的读入向量。

聚类算法中的很多操作属于计算密集型，这意味着这些操作，如向量序列化、反序列化、距离计算等都会使CPU满负荷运行。另一方面，有些操作属于I/O密集型，比如通过网络将中心向量传到每个Reducer。为提高聚类性能，需要理解解决上述两类性能瓶颈的一些技巧。下面将会看到，Mahout中的多个参数是如何基于输入数据的类型产生CPU、磁盘或网络瓶颈的。

为参考起见，图11-3列出了k-means聚类算法的原理示意图。其他像模糊k-means、狄利克雷及LDA聚类算法的架构都和k-means类似，因此对k-means的优化也适用于这些算法。

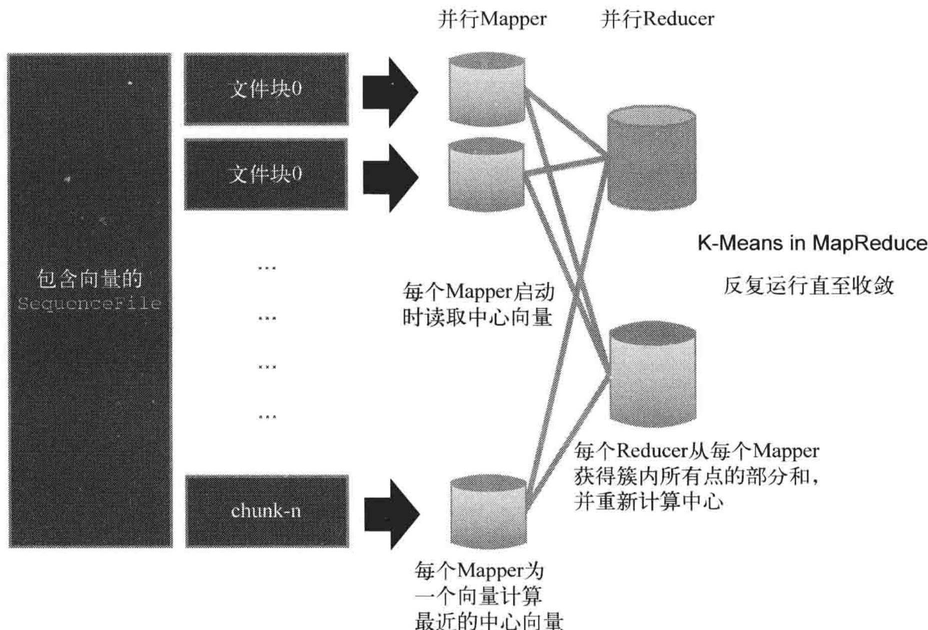


图11-3 以MapReduce作业方式迭代运行的k-means聚类原理示意图

聚类性能是输入数据的某个函数。为调节性能，需要分析输入的不同分布并确定在使用某些聚类参数时可能遇到的性能缺陷。下面我们首先看看如何减少CPU瓶颈甚至在某些情况下完全避免这种瓶颈。

11.2.1 在计算密集型操作中避免性能缺陷

当使用频繁的函数开始变慢时，CPU的性能会下降。在聚类中，距离计算是计算密集型操作，因此该计算越快，整个作业也就越快。当跟踪聚类中的CPU相关性能时，必须记住如下原则。

1. 采用合适的向量表示

在Mahout的所有3个Vector类中，DenseVector通常是最快的一个。可以快速访问该向量中的任意一个元素，也可以高效地顺序访问所有元素。但是如果用DenseVector表示稀疏向量，则会导致十分严重的性能问题。当用DenseVector表示一个存在很多零元素的向量时，会浪费很多存储的空间，而这些空间在稀疏表示下是根本不需要的。这也意味着很多数据做了没有必要的序列化处理，并且在希望忽略零元素的环境下遍历了很多零元素。

举个例子，比如你在对十分稀疏的文本向量数据（非零元素大概占整个向量维度的百分之一）进行聚类。那么在本地系统上运行时，使用SpareVector的聚类速度会是使用DenseVector的两倍，而在分布式系统上运行，前者要比后者快10倍。分布式聚类之所以会来带来额外的性能提高是由于用DenseVector表示时传输了不需要传输的0字节。

因此，通常最好将数据表示成SparseVector，这是因为即使有些稀疏，SparseVector表示也能大幅度节省存储、反序列化和网络传输带来的开销。

2. 使用更快的距离测度方法

聚类算法频繁计算距离，因此实现快速的距离计算相当关键。距离计算速度的提高会直接影响算法整体性能。

如果你在实现自己的距离测度方法，请遵从下面的最佳实践经验。

- 避免复制或者实例化一个新的Vector对象。Vector是重量级的Java对象，对它们进行复制会严重损害性能。
- 如果距离测度只需要非零元素，那么就应该避免对所有元素遍历。此时使用Vector.iterateNonZero()迭代器而不是Vector.iterator()。
- 使用Vector.assign()或Vector.aggregate()方法来高效遍历和修改向量。通过附件B可以了解更多关于向量的知识。

3. 根据距离计算使用SparseVector类型

有两种稀疏向量的实现，最好使用适合距离计算的实现。RandomAccessSparseVector擅长于随机查找，而SequentialAccessSparseVector擅长于快速顺序访问。

例如，计算余弦相似度需要很多向量点积运算，这需要在两个向量上依次遍历元素。实现代码需要将两个向量匹配位置上的值相乘。很自然地，对于这种距离计算中的顺序访问模式SequentialAccessSparseVector是最理想的，此时它会远远快于RandomAccessSparse-Vector。将所有文档向量保存为顺序格式会给距离计算带来巨大提升从而提高聚类的整体性能。

提示 Mahout utils包中有一个称为VectorBenchmarks的工具类。它在密集型、随机访问型和顺序访问型向量上使用不同类型的向量运算从而对比它们的速度。如果你定制的距离测度方法需要很多某种类型的向量运算，那么就可以使用该工具类来寻找某个稀疏水平上执行该运算最快的向量类型。在mahou-utils模块的org.apache.mahout/benchmark包中可以找到这个工具。

11.2.2　在I/O密集型操作中避免性能缺陷

通常影响I/O性能最大的是程序读取和写入的数据量。在Hadoop作业下，这些读/写操作大部分是顺序的。减少写入的字节量能够极大地提高整体的聚类性能。为降低I/O瓶颈，必须牢记如下的几条原则。

1. 使用合适的向量表示

这一点很明显。正如前面解释的那样，你永远不要将稀疏向量保存成`DenseVector`对象。这样做会使磁盘存储量极度膨胀从而导致聚类算法的磁盘和网络I/O瓶颈。

2. 使用HDFS副本

HDFS上的任一文件默认情况下都会在集群的不同节点上保存有三个副本。这样做能够防止某个节点上的副本丢失，另外它还能提高性能。如果数据只存放在一个节点上，那么所有需要这份数据的的`Mapper`和`Reducer`都会访问这个节点。这就会造成访问瓶颈。副本能够允许HDFS在不同服务器上保存文件块，这样就可以让Hadoop选择计算的初始化节点，从而能够解除上述网络I/O瓶颈。继续增加副本数目潜在上能够进一步解除上述瓶颈，但是这也意味着需要更多的HDFS存储开销。只有在由于副本数目不够导致瓶颈的情况下才考虑这样做。

3. 减少簇的数目

在聚类中，簇通常表示为很大的密集向量，每个向量都消耗相当大的存储空间。如果聚类作业试图寻找的簇数目（k）较多，那么这些向量就会通过网络从`Mapper`传输给`Reducer`。如果k较小，那么就可以减少网络I/O的开销。同时，这样也会减少k-means和模糊k-means算法中距离计算的开销，而原来每个簇中心的距离计算开销都会按磁盘上点的数目来增加。

当必须要将数据聚成较多数目的簇时，可以将数据放在Hadoop上并增加更多的机器来让它自己扩展。但是可以通过使用一个两步的批聚类方法将距离计算和聚类时间减少一个数量级。这种做法能够加快极大规模数据集如维基百科的聚类速度。

还有一种做法，我们可以探索更好的增量式聚类方法来减少聚类算法需要检查的数据量。这种做法有助于在线聚合器（如新闻网站），当新闻报道到来时自动将它们加到某个簇中。下一节当中将会解释上述两种做法。

11.3　批聚类及在线聚类

我们重新回到第9章提到的AllMyNew.com在线新闻门户网站。我们发现该网站的相关新闻功能的实现可以看成是一个简单的聚类问题。但是聚类却得不到我们想要的最终输出结果。假设该门户有大约100万篇新闻报道。我们要做的是产生最多100篇文档构成的簇，否则相关文章的列表太大不可用。但这也意味着需要产生10 000个簇。

这不是解决上述问题的好办法。尽管Hadoop可以允许我们向上扩展来完成上述任务，但是会浪费大量的CPU和磁盘，而这些都是需要花钱的。一个更好的办法是将所有文章划分成100个大簇，然后，对每个大约10 000篇文章的簇，进一步将它聚成100个更小的簇，这个过程如图11-4所示。

No. 11

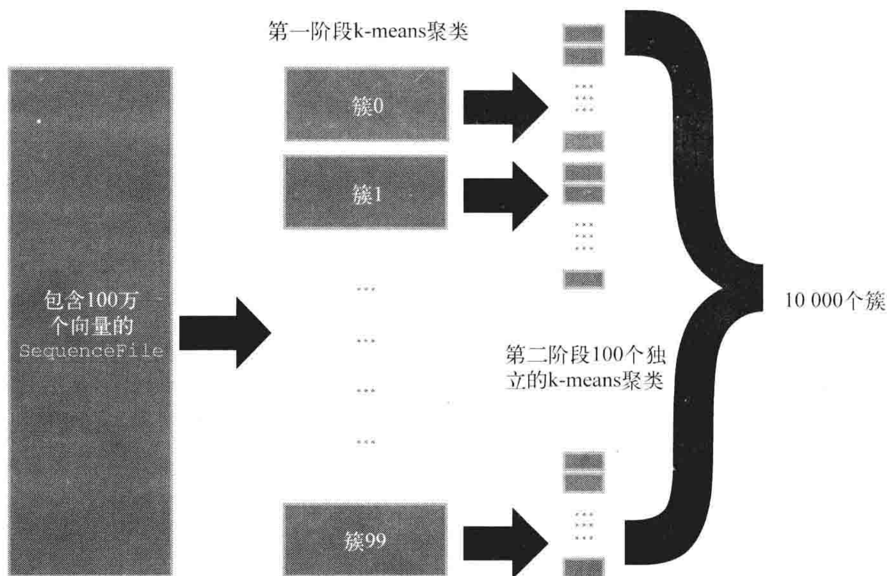


图11-4 采用两步方式来应用k-means算法。一个拥有10 000个簇的单个大MapReduce过程将有N×106×104次距离计算，其中N是迭代的次数。另一方面，将整个问题层次化为两层之后只有N1×106×102+N2×104×102×102次距离计算，整个计算次数差不多减少了100倍

在过去的示例中，我们一次性的对所有文章聚类，这叫批聚类（batch clustering）或离线聚类（offline clustering）。但是随着新闻门户中新文章的不断到达，这些新文章也需要被聚类。如果只是因为这些新文章就要重新反复运行批聚类作业的话，那代价就太昂贵了。即使这些批聚类作业每个小时运行一遍，那在一个小时内，我们还是无法识别与新文章相关的文章。这显然不是我们想要的结果。

下面介绍在线聚类（online clustering）技术，它的目标是在对已有对象聚类后对新对象高效聚类。

11.3.1 案例分析：在线新闻聚类

这里的在线聚类并非真的在线或者真的即时聚类。有一些对数据流直接进行聚类的算法，但是它们在扩展时会有问题。而我们考虑的是下面的技术。

- 同前面讨论的那样，我们对100万篇文章聚类，并保存所有簇的中心。
- 对每篇新文档，我们周期性地采用canopy聚类方法将它分配到离它最近的簇中，计算时使用一个非常小的距离阈值。这可以保证和先前主题有关的文章会和该主题关联并立即显示在网站上。这些归入已有簇的文章会被从新文档列表中删除。
- 所有剩余的不能归入已有簇的文章形成新的多个canopy，这些canopy代表了新闻中已经出现但是与过去的任何文章都不匹配或匹配度很小的主题。

- 利用这些新canopy的中心，对与已有簇不相关的文章进行聚类，并将这些临时的簇中心加入到中心列表中。
- 不太频繁地执行完整的批聚类算法来对全部文档集合进行聚类。这样做的时候，将所有已有的簇中心作为算法输入从而聚类算法可以更快地收敛。

这里的关键是确保增量式canopy聚类能够尽可能快地完成，从而用户能够在新闻报道发布数分钟之内就能看到相关文档簇。通过一个专用节点很容易实现这一点。另一种做法是将报道的发布时间延迟几分钟直到相关文档簇产生为止。

上面这个例子表明，某个解决方案可能会很容易耗尽所有的资源，能扩展也不意味着就应该使用大量资源。我们也可以在一个专用的数千节点的集群上运行完整的聚类算法，但是这样做会浪费钱。稍微用点聪明才智就可以得到一个更便宜也更快的解决方案。

11.3.2 案例分析：对维基百科文章聚类

维基百科的文章每晚都会导出为XML文件并可从如下链接下载：http://dumps.wikimedia.org/enwiki/。这是一个神奇的数据集。由于是人工编辑和组织的，很多文章在维基百科中仍然没有分组或者进行了错误分组。你可以尝试在这些文章上运行聚类算法来获取相关文档，这没准还能帮助编辑人员在维基百科网站上将相关文章分组呢！

在尝试对文章聚类之前，需要从XML格式的文件中提取文档和向量。幸运的是，Mahout中有一个维基百科数据集的创建类，它可以读入XML格式并以`SequenceFile`格式输出每篇单独的文章。对于本实验的目的而言，我们将提取科技分类下的所有文档。

首先，从前面提到的网站下来最新的pages-articles.xml.bz2文件。将该文件解压到本地目录然后上传到HDFS。假设Hadoop的环境变量已经设置好，运行如下的维基百科提取命令：

```
echo "science" > categories.txt
bin/mahout seqwiki -c categories.txt -i articles.xml \
  -o wikipedia-seqfiles -e
```

上述命令会从XML文件中提取那些维基百科分类与*science*精确匹配的文章然后将这些文章写到`SequenceFile`中。对于该输入可以运行`SparseVectorsFromSequenceFiles`作业来创建维基百科的TF-IDF向量。然后，在其上可以运行任意聚类算法。整个过程会持续一段时间。

一旦成功运行聚类算法之后，你可能想通过实验来观察比如向量表示对聚类算法的影响。在词典向量化工具中使用-seq标志将向量创建为`SequentialAccessSparseVector`。这将会比默认创建的`RandomAccessSparseVector`要快。

一旦有足够的信心对Wikipedia样本集聚类，那么通过如下命令提取所有数据并对它向量化：

```
bin/mahout seqwiki -all -i articles.xml -o wikipedia-seqfiles
```

如果不访问一个强大的Hadoop集群，那么上述命令会在单机上执行大约一整天。另一种做法是在云上运行，具体做法将在下面看到。

在亚马逊弹性MapReduce下运行维基百科聚类

一种比较便宜的实现Mahout的方法是从类似亚马逊弹性MapReduce（Amazon Elastic MapReduce，EMR）云服务中购买Hadoop集群时间。本书的6.6节给出了如何访问该资源的快速方法。

如果你觉得亚马逊弹性MapReduce服务还不错的话，可以像下面这样运行Mahout算法。

(1) 将Mahout作业文件（mahout-examples-0.5-job.jar）上载到一个Amazon S3 bucket。

(2) JAR文件出填上*-job.jar文件的目录。

(3) 为需要运行的算法的驱动类指定完整的类名及其参数，该类的类名或许是`org.apache.mahout.clustering.kmeans.KMeansDriver`。

(4) 指定存储在S3 bucket中的维基百科向量文件的完整输入路径以及簇的完整输出路径。

(5) 指定其他需要的MapReduce参数，比如机器的数目。记住，每个计算单元按小时计费，因此大规模集群很快就会花费很多钱。

(6) 运行作业。

你可以通过上传`SequenceFile`并运行`SparseVectorsFromSequenceFile`类，按照相同流程来直接使用EMR服务生成向量。

在写这本书的时候，用了8节点的集群大概1个小时来将维基百科的`SequenceFile`转换成向量。聚类的时间高度依赖于初始的中心、距离计算方法和簇的数目，当然也取决于计算所用的机器数目。要永远记住，计算不是免费的。

11.4 小结

在这一章里，我们简单看了看Hadoop集群以及如何在它上面运行聚类作业。我们还讨论了Mahout启动脚本及其两种操作模式。

我们讨论了聚类性能中的多个问题，并分析了可以缓解聚类时CPU和I/O瓶颈的多项技术。正确的向量表示和优化的距离算法是高性能聚类应用的关键。

我们也将上述知识用于一个假想的新闻聚合门户——AllMyNews.com的伪在线聚类引擎的实现中。加入一点点精心设计之后，通过使用多种技术的组合，整个大的聚类问题被划分成多个简单快速的小问题，这样系统中就可以达到近似实时的相关文档聚类效果。为突出Mahout的聚类扩展能力，我们试图对一个当前最大的公开数据集即英文的维基百科进行聚类。由于整个过程很慢，我们快速讨论了如何通过使用亚马逊弹性MapReduce服务云来进行多节点聚类的过程。

通过上述案例，可以学到在Hadoop集群上运行Mahout及对其进行扩展的知识。本书到达本章这一部分的旅程并不简单。我们一开始在第8章介绍了在平面上对点进行聚类的基本知识，那里我们学到了向量及距离计算方法。然后我们尝试了Mahout中的多种算法并试图将它们应用于实际案例中。我们从小规模非分布式计算逐渐过渡到大规模分布式计算。同时，你也学会了如何调整算法的速度和质量。下一章，即第12章，你将把学到的知识应用到实际案例中。

第 12 章 聚类的实际应用

12

本章概要

- ❑ 将有相同兴趣的Twitter用户聚类到一起
- ❑ 利用聚类为Last.Fm上的艺术家推荐标签
- ❑ 为网站构建相关帖子的功能

或许你拿起这本书是想学习和理解聚类是如何解决实际问题。迄今为止，我们主要集中探讨了路透社新闻数据集上的聚类问题，这个数据集大约有20 000篇文档，每篇文档大约有1000到2000词。该数据集对于Mahout来说并不足以构成挑战，无法显示出Mahout的扩展能力。本章当中，我们主要使用聚类算法来解决三个大得多的数据集上的聚类问题。

首先，我们尝试使用来自Twitter（http://twitter.com）的公开推文，通过聚类寻找推文内容相似的用户。其次，我们会考察一份来自Last.fm（http://last.fm）的数据集，该网站是一个流行的互联网电台，我们利用这些数据来产生相关的标签。最后，我们使用一个著名技术论坛网站Stack Overflow（http://stackoverflow.com）导出的全部数据，该数据包括500 000个问题和200 000个用户。我们利用该数据来实现网站的相关特性功能。

第一个问题是通过对Twitter的推文聚类来找到相似用户。

12.1 发现 Twitter 上的相似用户

Twitter是一个提供微博服务的社交网站，用户可以公开发布简短的消息，称为推文（tweet）。这些推文最多包含140个字符。推文一旦发布，它就会出现在该推文作者的所有粉丝的订阅信息流中，并且该推文在Twitter网站公开可见。这些推文有时会包含某些特殊格式的关键词，例如`#Obama`，或者直接通过@符号引入其他用户的名字，如`@SeanOwen`。

由于Twitter新的服务条款不允许公开发布数据集，因此需要你自己来准备这些数据。在本书的源码当中，有一个称为`TwitterDownloader`的类能够从Twitter上获取100 000条推文（可以在源码中修改这个数字）并写到一个文件中，该文件可以用于后续的分析过程。获得100 000条推文大约需要两小时。

注意 要运行上述程序，必须要在https://dev.twitter.com/apps注册你的应用，并从Twitter获得一个认证密钥，然后按照源码中的描述将该密钥值写到程序中。在运行时不要忘了加上-Dfile.encoding=UTF-8选项，否则会丢失所有非ASCII的字符。

12.1.1 数据预处理及特征加权

我们需要对上述数据进行预处理，将其换成Mahout支持的格式。首先需要将其转换成可以被词典向量化工具读取的SequenceFile格式。然后，再利用TF-IDF权重计算，将其转换成向量。

词典向量化工具期望的输入包含一个Text类型的键和值。这里的键将是Twitter的用户名，而值将是该用户发表的所有推文的拼接。

MapRuduce作业很适合完成上述输入的准备工作。Mapper读入数据中的每一行，即通过制表符分隔的用户名、时间戳和推文。然后将用户名作为键，推文作为输出。Reducer接收来自同一用户的所有推文，然后把它们连接成一个字符串，并将它作为值输出，此时键仍然为用户名。上述结果会被写入到SequenceFile文件中。

上述过程中的Mapper、Reducer、Configuration类及在Hadoop集群中调用它们的一个作业均列在代码清单12-1和代码清单12-2中。其中MapReduce类是一个通用的按字段分组的Mapper。

代码清单12-1 按字段分组的Mapper

```
public class ByKeyMapper extends Mapper<LongWritable,Text,Text,Text> {
  private Pattern splitter = Pattern.compile("\t");
  private int selectedField = 1; // tweet
  private int groupByField = 0; // username

  @Override
  protected void map(LongWritable key, Text value,
 Context context) throws IOException,
InterruptedException {
    String[] fields = splitter.split(value.toString());      <- 利用正则表达式对行进行分割
    if (fields.length - 1 < selectedField ||
        fields.length - 1 < groupByField) {
      context.getCounter(
          "Map", "LinesWithErrors").increment(1);          <- 对行错误计数
      return;
    }

    String oKey = fields[groupByField];
    String oValue = fields[selectedField];
    context.write(new Text(oKey), new Text(oValue));      <- 输出用户名和推文

  }

}
```

代码清单12-2　按字段分组的Reducer

```
public class ByKeyReducer extends Reducer<Text,Text,Text,Text> {
  @Override
  protected void reduce(Text key,
                        Iterable<Text> values,
                        Context context) {
    StringBuilder output = new StringBuilder();
    for (Text value : values) {                          <-- 读入用户的推文
      output.append(value.toString()).append(" ");       <-- 产生拼接的字符串
    }
    context.write(key, new Text(
        output.toString().trim()));                      <-- 输出拼接后的字符串
  }
}
```

一旦作业运行，用户的推文就会被拼接成文档大小的字符串并写入到SequenceFile中。然后，该输出结果会被转换成TF-IDF向量。在运行基于词典的向量化工具之前，我们值得花一些时间来更好地了解Twitter数据并识别一些简单有用的改进点，这些点有助于特征选择和加权。

12.1.2　避免特征选择中的常见陷阱

TF-IDF是一种非常有效的特征权重计算方法，前面我们已经用过多次。如果将它应用于单独的一条推文，由于文本最多只有140个字符或者说大约总共25个单词，所以大部分单词的频率都是1。我们感兴趣的是将推文类似的用户聚类，因此我们创建的数据集中包含用户所有的推文而不只是一条推文。这样的话使用TD-IDF会更有意义，聚类也会更有意义。

直接应用TF-IDF效果会一般。如果将推文中常见的停用词去掉效果会更好。我们使用词典向量化工具并将最大的文档频率比率参数设置为一个较低的值，比如50%。这会去掉所有出现在一半以上文档中的单词。我们还使用该工具中的搭配特征来生成Twitter数据中的二元组（bigram）并将它们也用作聚类特征。

但是这里存在一个问题。推文是一种写作上比较随便的消息，而不像路透社新闻语料中的文本那样经过精心编辑加工。因此，推文中包含拼写错误、省略、俚语和其他的文本变化方式。三个用户写的三条如下推文之间由于没有任何公共词，因此永远无法将其聚类到一起：

```
HappyDad7: "Just had a refreshing bottle of Loke"
BabyBoy2010: "I love Loca Kola"
Cool_Dude9: "Dude me misssing LLokkkeee!!!!"
```

为解决上述问题，可以通过语音过滤器（phonetic filter）来发现上述拼写之间的关系。这些语音过滤器将单词还原为一个能够接近单词大致发音的基本形式。比如，Loke和Loca可能都被还原成LK，而单词refreshing可能被还原成RFRX。上述过程至少有三种著名的实现方法：Soudex、Metaphone和Double Metaphone。所有方法都实现于Apache Commons Codec库中[①]。下面我们使用

① Apache Commons Codec包的层次结构参见http://commons.apache.org/codec/apidocs/org/apache/commons/codec/language/package-tree.html。

DoubleMetaphone实现方法来将每个词转换成其基本发音的形式。下面的代码清单给出了使用DoubleMetaphone的一个Lucence Analyzer实现示例程序。

代码清单12-3 为推文优化的Lucene Analyzer类

```
public class TwitterAnalyzer extends Analyzer {
  private DoubleMetaphone filter = new DoubleMetaphone();    // 初始化编码器
  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
    final TokenStream result = new PorterStemFilter(new StopFilter(
        true, new StandardTokenizer(Version.LUCENE_CURRENT, reader),
        StandardAnalyzer.STOP_WORDS_SET));

    TermAttribute termAtt = (TermAttribute) result
        .addAttribute(TermAttribute.class);
    StringBuilder buf = new StringBuilder();
    try {
      while (result.incrementToken()) {
        String word = new String(termAtt.termBuffer(), 0, termAtt
            .termLength());
        buf.append(filter.encode(word)).append(" ");    // 建立发音词根
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    return new WhitespaceTokenizer(new StringReader(buf.toString()));
  }
}
```

注意 DoubleMetaphone过滤器只能应用于英语文本。对于其他语言可能需要寻找其他的过滤器。当从Twitter下载推文时，数据集中也会包含其他语言的推文。在这个例子中我们忽略了其他语言带来的结果损失。

利用HDFS复制命令将12.1.1节产生的SequenceFile复制到集群上，一旦拷上去之后，就可以在Twitter SequenceFile上运行DictionaryVectorizer命令：

```
bin/mahout seq2sparse -ng 2 -ml 20 -s 3 -x 50 -ow \
-i mia/twitter_seqfiles/ -o mia/twitter-vectors -n 2 \
-a mia.clustering.ch12.TwitterAnalyzer -seq
```

要确保运行上述命令时TwitterAnalyzer类处于Java的CLASSPATH下，这一点很重要。向量化工具会花一些时间对推文进行词条化处理并产生完整的二元组集合。向量化表示将比原始输入所占的空间小。

我们并不知道聚类要产生的簇的个数。由于这里产生推文的独立用户不足100 000个，因此可以每100个用户创建一个簇，从而得到大约1000个簇。使用刚才产生的向量运行k-means算法。输出结果中得到的一个簇的例子如下所示：

```
排名靠前的词项：
        KMK                      =>   7.609467840194702
        APKM                     =>  7.5810538053512575
        TP                       =>   6.775630271434784
        KMPN                     =>   5.795418488979339
        MRFL                     =>  4.9811420202255245
        KSTS                     =>   4.082704496383667
        NFLX                     =>   4.005469512939453
        PTL                      =>  3.9200561702251435
        N                        =>  3.7717770755290987
        A                        =>  3.4771755397319795
```

这里的特征KMK是DoubleMetaphone过滤器对comic、comics、komic等之类的词作用后的输出结果。为得到原始词，不使用上述过滤器处理或直接简单地使用StandardAnalyzer来对向量处理并聚类即可。

可以将最大的文档频率参数设置为50%和20%分别运行两次聚类算法。这样做以后，有关主题comics的簇中排名靠前的词条如下面所示：

```
排名靠前的词条，-maxDFPercent=50
        comic                    =>   9.793121272867376
        comics                   =>   6.115341078151356
        con                      =>   5.015090566692931
        http                     =>   4.613376768249454
        i                        =>   4.255820257194115
        sdcc                     =>   3.927590843402978
        you                      =>   3.635386448917967
        rt                       =>  3.0831371437419546
        my                       =>  2.9564724853544524
        webcomics                =>   2.916910980686997

排名靠前的词项，-maxDFPercent=20
        webcomics                =>  11.411304909842356
        comic                    =>  10.651778670719692
        comics                   =>  10.252182320186071
        webcomic                 =>   7.813077817644392
        con                      =>   5.867863954816546
        http                     =>   4.937416185651506
        companymancomic          =>   4.899269049508231
        spudcomics               =>   4.543261228288923
        rt                       =>   4.149479137148176
        addanaccity              =>   4.056342724391392
```

上述两个输出结果表明了在像推文这样的带噪音的数据中进行特征选择的重要性。当最大文档频率参数设置为文档数的50%时，有一些像i、you、my、http和rt的特征存在。这些特征占据了距离计算中的主要地位，因此它们在簇中心里具有较高的权重。降低上述阈值会去掉大部分这样的词。但是像http和rt之类的词仍然出现在下面那个簇中，理想上它们应该被去掉。但是如果设置非常低的阈值可能会有风险，因为此时可能会丢失一些十分重要的特征。一种更好的做法是手工建立一个词汇列表然后利用Lucene StopFilter去掉诸如bit.ly、http、tinyurl.com、rt之类的单词。

上面只利用单词来计算用户的相似度，然而也可以利用用户的交互信息来推导出它们之间的相似度。例如，如果用户A转发（含转发或重述）了用户B的推文，那么就可以将B看成是A用户

向量的一个特征，该特征的权重是A转发B推文的次数。对按照用户兴趣聚类来说，这可能是的一个好特征。

或者，也可以将推文中用户之间互相提及的次数作为特征的权重，但是如果这样做的话，该特征可能会对聚类的结果产生负面影响。该特征可能对于将用户聚成讨论社区非常有用，但是对于聚成兴趣相投的用户群可能并不好。对某个问题最好的特征并不一定适用于其他问题。

通过聚类，可以了解不同的技术博客作者如何聚到一起，而不同的朋友群又如何聚到一起。这个问题还是留给读者自己去探查和实验吧。下面，我们会考察另一个有趣的问题：为一个Web 2.0电台开发一个标签推荐引擎。

12.2 为 Last.fm 上的艺术家推荐标签

Last.fm是一个流行的互联网音乐电台网站。Last.fm有很多功能，我们感兴趣的是用户能够用词来对歌曲或艺术家打标签。例如，Green Day乐队可以打上punk、rock或者greenday之类的标签。而The Corrs乐队可以使用violin和Celtic来作为标签。

下面我们试图通过聚类来实现一个相关标签推荐的功能。这可以辅助用户来打标签：用户输入一个标签之后，系统可以推荐一些在某种意义上常常与该标签共现的其他标签。比如，在用户对歌曲打上标签punk后，系统可能会推荐标签rock。接下来我们会探讨两种实现方法：第一种基于简单的共现信息来实现，第二种通过某种形式的聚类来实现。

我们的输入数据集记录了每个艺术家采用某个单词作为标签的次数。该数据集可以从地址http://musicmachinery.com/2010/11/10/lastfm-artisttags2007/下载。它包含了Last.fm音乐听众在那段时间对20 000个艺术家进行标注所使用的频率最高100个标签的原始标记次数。数据集的格式如下：

```
artist-id<sep>artist-name<sep>tag-name<sep>count
```

每行包含一个艺术家ID及其姓名、一个标签名和该标签标记该艺术家的次数。

12.2.1 利用共现信息进行标签推荐

我们可以利用共现信息，或者说两个标签同时对某个艺术家进行标记的次数作为相似计算的基础。在前面的第6章我们讨论过共现信息，那里计算的是物品的共现信息。有了相似度计算方法之后，就可以进行聚类。

对每个标签而言，可以计算它和其他标签的共现次数然后将共现次数最高的那些标签作为推荐结果。这里只有一个问题：一个频繁出现的标签，如awesome，会同时和其他标签共现，因此即使它作用不大，它仍然会作为很多标签的推荐标签。从这个角度来看，我们必须要去掉那些高频标签。

然而，采用上述共现技术来推荐标签，那么那些间接关联的标签永远不会被推荐。也就是说，如果A和B共现，B和C共现，当用户添加标签A时，上述基于共现信息的方法永远不会推荐C。一个变通的方案就是将间接的标签也加入到推荐标签列表中并对它们进行重新排序。

12

但是对于解决上述问题来说，聚类其实已经实现了这个功能。如果标签之间的相似度在阈值之内，不管它们是否共现都会被聚成一类。

我们可以将上述问题建模为基于艺术家的聚类问题，其中艺术家是文档，而特征是艺术家收到的标签。另一种替代方案是，将标签看成文档而将艺术家看成文档中的词，并将它们共现的次数作为词频。这是分析同一问题的两个不同角度，可以基于图12-1中的二分图（bipartite graph）进行建模。

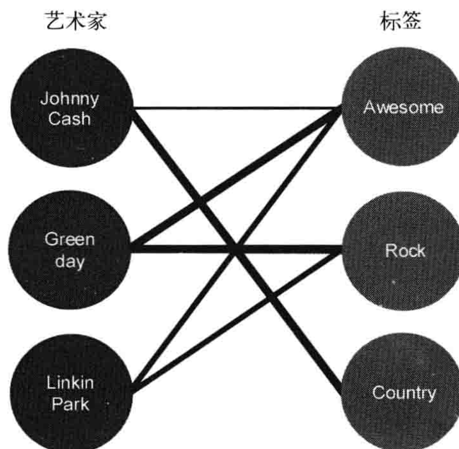


图12-1　一个典型的二分图（一个可以将节点分割成两部分的图）。这里一类节点代表艺术家，另一类代表标签。边上的权重等于标签应用于艺术家的次数。在推荐系统中，节点由用户和物品构成，边上的权重是用户标的星级。推荐和聚类算法之间的密切程度比人们想象的高

Last.fm数据集看上去和上面的表示类似，可以直接转换成向量。我们建立一个简单的向量化工具，首先生成一部艺术家的词典，然后利用该词典将艺术家转换成MapReduce形式的整数向量。我们本可以使用字符串作为ID，但是Mahout `Vector`形式只支持整数维度。当看到将Last.fm艺术家转换为标签向量的代码时，这一原因便一目了然。

12.2.2　构建Last.fm艺术家词典

为了生成Last.fm数据集的特征向量，我们部署两个MapReduce作业。第一个作业以词典的形式生成独立的艺术家列表，第二个作业利用生成的词典来产生向量。词典生成的`Mapper`和`Reducer`类的代码在下面的代码清单12-4、代码清单12-5和代码清单12-6中给出。

代码清单12-4　从数据集输出艺术家

```
public class DictionaryMapper extends
Mapper<LongWritable,Text,Text, IntWritable> {
  private Pattern splitter;
```

```
  @Override
  protected void map(LongWritable key, Text line, Context context)
       throws IOException,
                 InterruptedException {
    String[] fields = splitter.split(line.toString());
    if (fields.length < 4) {
      context.getCounter("Map", "LinesWithErrors").increment(1);
      return;
    }
    String artist = fields[1];          <-- 从分割行中选择艺术家

    context.write(new Text(artist), new IntWritable(0));   <-- 将艺术家作为键输出
  }
  @Override
  protected void setup(Context context) throws IOException,
                                      InterruptedException {
    super.setup(context);
    splitter = Pattern.compile("<sep>");
  }
}
```

上述Mapper接受一行文本作为输入然后将艺术家作为键输出，此时其对应的值为空。该键–值对会传送给Reducer。

代码清单12-5　按艺术家分组并输出某个艺术家的唯一键–值对

```
public class DictionaryReducer extends
Reducer<Text,IntWritable,Text,IntWritable> {
  @Override
  protected void reduce(Text artist,
                        Iterable<IntWritable> values,
                        Context context) throws IOException,
                                  InterruptedException {
    context.write(artist, new IntWritable(0));   <-- 只选择键
  }
}
```

Mapper输出艺术家，而Reducer只输出唯一的艺术家并丢弃空的字符串值。一旦建立了词典，就可以将其读入到一个HashMap中，其中每个词都会映射为一个唯一的索引值，该过程如下面的代码清单所示，实现时是串行执行的。

代码清单12-6　使用唯一的艺术家姓名并为每个艺术家创建一个唯一的整数ID

```
  ...
    Map<String,Integer> dictionary = new HashMap<String,Integer>();
    FileSystem fs = FileSystem.get(dictionaryPath.toUri(), conf);
    FileStatus[] outputFiles = fs.globStatus(
new Path(dictionaryPath, "part-*"));        <-- 选择所有的词典文件
    int i = 0;
    for (FileStatus fileStatus : outputFiles) {
      Path path = fileStatus.getPath();
      SequenceFile.Reader reader = new SequenceFile.Reader(fs, path,
          conf);
      Text key = new Text();
      Text value = new Text();
```

```
      while (reader.next(key, value)) {
        dictionary.put(key.toString(),          将唯一的标签放入词典中
            Integer.valueOf(i++));
      }
    }
  ...
```

这个HashMap会被序列化并保存在Hadoop的Configuration对象中。然后就进入下一个Mapper，该Mapper利用这部词典将字符串标签转换成整数维ID。下面我们会看到这一点。

12.2.3　将Last.fm标签转换成以艺术家为特征的向量

利用上一节生成的词典，可以建立标签向量而后对它们进行聚类。向量化过程只是个简单的MapReduce作业。Mapper选择艺术家的整数特征ID然后建立单个特征的向量。这些一维的部分向量会传输给Reducer，后者会将这些向量简单地进行联结，生成一个完整的向量。上述过程如代码清单12-7所示。

代码清单12-7　利用艺术家的整数ID映射将标签转换成向量

```
public class VectorMapper extends
    Mapper<LongWritable,Text,Text,VectorWritable> {
  private Pattern splitter;
  private VectorWritable writer;
  private Map<String,Integer> dictionary = new HashMap<String,Integer>();

  @Override
  protected void map(LongWritable key, Text value,
      Context context) throws IOException,
                              InterruptedException {
    String[] fields = splitter.split(value.toString());
    if (fields.length < 4) {
      context.getCounter("Map", "LinesWithErrors").increment(1);
      return;
    }
    String artist = fields[1];                     从分割文本中选择字段
    String tag = fields[2];
    double weight = Double.parseDouble(fields[3]);
                                                          创建向量
    NamedVector vector = new NamedVector(
        new SequentialAccessSparseVector(dictionary.size()), tag);

    vector.set(dictionary.get(artist), weight);        将ID和权重写入向量
    writer.set(vector);

    context.write(new Text(tag), writer);          输出部分向量
  }

  @Override
  protected void setup(
      Context context) throws IOException,              Mapper之前的设置
                              InterruptedException {
    super.setup(context);

    Configuration conf = context.getConfiguration();
```

```
    DefaultStringifier<Map<String,Integer>> mapStringifier
        = new DefaultStringifier<Map<String,Integer>>(
              conf, GenericsUtil.getClass(dictionary));      ← 反序列化HashMap
    dictionary = mapStringifier.fromString(conf.get("dictionary"));
    splitter = Pattern.compile("<sep>");
     writer = new VectorWritable();
  }
}
```

我们使用SequentialAccessSparseVector作为标签向量的表示形式。k-means算法会对向量元素进行多次顺序的遍历，上述表示最适合于这种访问模式。利用Mapper输出的部分向量，Reducer通过简单的拼接得到一个完整的向量。具体过程如下所示。

代码清单12-8　组合部分标签向量并累加成完整向量

```
public class VectorReducer extends
    Reducer<Text,VectorWritable,Text,VectorWritable> {
  private VectorWritable writer = new VectorWritable();

  protected void reduce(Text tag,
                        Iterable<VectorWritable> values,
                        Context context) throws IOException,
                                InterruptedException {
    Vector vector = null;
    for (VectorWritable partialVector : values) {
      if (vector == null) {
        vector = partialVector.get().like();              ← 初始化空向量
      }
      partialVector.get().addTo(vector);                ← 将部分向量合并成完整向量
    }

    NamedVector namedVector = new NamedVector(vector, tag
        .toString());
    writer.set(namedVector);

    context.write(tag, writer);                     ← 输出完整向量
  }
}
```

VectorReducer将部分向量合并成一个完整向量。现在Last.fm的艺术家向量已经就绪。下一步就是运行k-means聚类算法来得到主题。

12.2.4　在Last.fm数据上运行k-means算法

Last.fm数据集的向量化过程会产生大概98 999个不同的标签。你可能想将这些标签聚成大概每个包含约30个标签的簇。

通过如下命令，我们尝试利用k-means算法来建立2000个标签簇：

```
bin/mahout kmeans -i mia/lastfm_vectors/ \
-o mia/lastfm_topics -c mia/lastfm_centroids -k 2000 -ow \
-dm org.apache.mahout.common.distance.CosineDistanceMeasure \
-cd 0.01 -x 20 -cl
```

在单台机器上，上述迭代过程大约需要30分钟。该过程属于计算密集型，因此在Hadoop集群中增加更多的工作机便能减少执行的时间。

一旦得到输出之后，可以利用clusterdump工具来查看这些标签簇。具体命令如下：

```
bin/mahout clusterdump -s mia/lastfm_topics/clusters-16 \
-d mia/lastfm_dict/part-r-00000-dict -dt sequencefile -n 10
```

下面是给定某个标签下排名靠前的艺术家：

```
排名靠前的词项:
        Shania Twain                  =>   1.126984126984127
        Garth Brooks                  =>   0.746031746031746
        Sara Evans                    =>  0.6031746031746031
        Lonestar                      =>  0.5238095238095238
        SHeDaisy                      => 0.47619047619047616
        Faith Hill                    =>  0.4126984126984127
        Miranda Lambert               => 0.36507936507936506
        Dixie Chicks                  => 0.36507936507936506
        Brad Paisley                  =>  0.3492063492063492
        Lee Ann Womack                =>  0.3492063492063492

排名靠前的词项:
        Explosions in the Sky         =>   55.19047619047619
        Joe Satriani                  =>   51.19047619047619
        Apocalyptica                  =>   50.95238095238095
        Steve Vai                     =>  43.666666666666664
        Mogwai                        =>   39.57142857142857
        Godspeed You! Black Emperor   =>   32.00100000000101
        Yann Tiersen                  =>  30.857142857142858
        Pelican                       =>  20.285714285714285
        Do Make Say Think             =>  16.904761904761905
        Liquid Tension Experiment     =>   16.61904761904762

排名靠前的词项:
        Justin Timberlake             =>   6.739130434782608
        Disturbed                     =>   0.391304347826087
        Timbaland                     => 0.34782608695652173
        Nelly Furtado                 => 0.30434782608695654
        Lustans Lakejer               =>  0.2608695652173913
        Michael Jackson               =>  0.2608695652173913
        Aaliyah                       => 0.21739130434782608
        Timbaland Magoo               => 0.21739130434782608
        Jonathan Davis                => 0.21739130434782608
        Sum 41                        => 0.21739130434782608
```

这些标签可以存储在数据库中，而后有需要时通过查询该数据库可以为任意标签进行推荐。

这个例子展示了这样一个技巧，即可以将数据集按照某种不同的视角进行转换以便进行聚类。下一节的例子会涉及更多的内容，考虑的是一个流行的问答网站Stack Overflow的公开数据集，该数据集比刚才讨论的数据集要大很多。

12.3 分析 Stack Overflow 数据集

Stack Overflow（http://stackoverflow.com）是一个在线的问答网站，该网站提供了各种各样的编程问题和解答。用户可以提出或解答问题，也可以对问题及答案进行投票以支持或反对。通过与网站的交互，用户也可以获得一些声望得分。比如，用户一旦收到某问题的答案的支持投票，那么就得10分。累积足够的分数之后，用户可以获得网站的额外特权。

感谢Stack Overflow，它们网站的所有数据都可以免费用于非商业研究。我们利用Stack Overflow的数据集，觉得其中一些可以看成聚类问题。比如，对问题进行聚类以找到相关问题，对用户进行聚类以发现相关用户，对相似答案聚类来保证答案更具可读性。

12.3.1 解析Stack Overflow数据集

stackoverflow.com及其姊妹网站（superuser.com、serverflow.com等，这些网站统称为Stack Exchange平台）的全部数据集打包成一个文件，可以从地址http://blog.stackoverflow.com/category/cc-wiki-dump下载。整个数据集的压缩包大概有3.5GB左右。所有数据集本质上极其类似，其中Stack Overflow是其中最大和最主要的部分。解压之后，Stack Overflow数据集大概5GB左右，这些数据被分割为多个XML文件。发帖、评论、用户、投票和论坛徽章都有各自独立的文件。

解析XML文件可能需要点技巧，特别是在Hadoop MapReduce作业中。Mahout为在MapReduce作业中读取XML提供了一个`org.apache.mahout.classifier.bayes.XMLInputFormat`类。它会正确检测到XML中的每个重复块并将它以输入记录方式传给`Mapper`。我们可以进一步通过任意XML解析库，以XML的方式解析该字符串。

`XMLInputFormat`类需要XML块的开始和结束标记。这可以通过在Hadoop的`Configuration`对象中设置如下键来实现：

```
...
conf.set("xmlinput.start", "<row Id=");
conf.set("xmlinput.end", " />");
...
conf.setInputFormat(XmlInputFormat.class);
...
```

在`Mapper`中，将键设置为`LongWritable`类型将值设置为`Text`类型，然后将整个XML块作为`Text`读入到值中，其中包括开始和结束符。

12.3.2 在Stack Overflow中发现聚类问题

Stack Overflow数据集包含一张用户表、一张问题表和一张答案表。它们之间存在关联，基于用户ID、问题ID和答案ID将有助于将数据转换成你所要的形式。对于该数据集有如下一些有趣的聚类问题：

1. 对发帖进行聚类以发现相关问题

识别与某个主题相关的问题十分有用。用户在浏览一个没有很好地解决其问题的帖子时，也

12

许可以在一些相关问题中找到更好的答案。这种寻找相似问题的过程就像前面提到的在路透社数据集上寻找相似文章。但这里的帖子比路透社新闻要短而比Twitter上的推文要长，因此TF-IDF权重计算方法应该可以得到较好的特征。我们最好也去除数据集中的高频词，因此这里将最大的文档频率阈值设为70%或更低。

在对上述数据进行向量化时面临的一个巨大挑战是缺乏一个Stack Overflow问题的好的词条化工具。很多问题和答案都包含来自不同编程语言的代码片段，而默认的`StandardAnalyzer`并未被设计成可以处理这类数据。因此需要编写解析器来处理代码中的括号和数组以及不同编程语言的奇怪格式。

除了只使用问题之外，还可以将问题和它们的答案及评论打包在一起产生更大的文档来得到更多的问题聚类特征。与Twitter不同，由于内容较大，因此这里的拼写错误不会对聚类的质量造成太大的影响。但是增加一个`DoubleMetaPhone`过滤器还是可以稍微提高一点聚类质量的。由于数据很多，因此k-means和模糊k-means都会产生类似的结果。只有使用LDA主题作为特征才可以得到更高质量的结果，但是在该数据集上运行LDA时的CPU消耗可能会高的离谱。

2. 对用户数据进行聚类以发现相似用户

假设你是一个长期使用JMS（Java Messaging Service，Java消息服务）API的开发人员，那么对你而言找到那些也使用JMS的用户十分有用。帮助用户形成这样的社区不仅可以提高网站的用户体验，还可以激发用户的参与度。与前面一样，这里可以通过聚类来计算出这种可能的社区。

对用户聚类需要用户的特征向量。这些特征可以是用户发的帖子或解答的内容，或者是用户和其他用户的交互信息。下面给出了向量的一些特征：

- 用户创建的问题或解答的内容，包括来自文本和代码片段的*n*元组（*n*-gram）；
- 对当前用户发的帖子进行回复或评论的其他用户。

可以只利用发帖的内容对用户聚类，也可以只利用共同的交互数目对用户聚类，或者两者同时使用。前面在对推文进行聚类时，只用到了内容信息。而利用交互特征来对用户聚类会是一个很好的实践体验。

在该模型中，用户是文档，其特征是其他与之有过交互的用户的ID，每个特征值反映的是用户之间的交互程度。该交互程度可以通过权重来计算。例如，如果用户2对用户1的发帖进行了评论，那么可以映射为权重10。但是如果用户2对用户1的发帖进行了投票，那么只会赋予权重2。如果是投反对票的话，那么可能会得到一个−10的权重，表示这两个用户交互的概率很低。考虑所有可能的交互行为并赋予相应权重。在用户1特征向量中的用户2特征的权重为它们之间所有交互权重的累加值。利用这些向量及特征，就可以尝试基于用户的交互模式对用户进行聚类并看看结果怎样。

12.4　小结

本章中，我们通过分析3个数据集Twitter、Last.fm和Stack Overflow考察了实际中的聚类应用。

一开始我们分析推文并试图找到推文类似的用户。我们对数据进行预处理，然后转换成向量，最后利用这些向量按照用户的推文相似度成功地对用户进行了聚类。由于采用了基于MapReduce的实现，Mahout很容易就能扩展并能处理一千万条推文构成的数据集。

我们利用了Last.fm标签数据集的一个小规模子集来对网站做标签推荐。我们将该问题建模成一个二分图网络，其中一类节点是音乐家另一类节点是用户产生的标签，边上的权重是它们共现的次数。我们基于标签进行了聚类。利用聚类的结果，很容易就能为网站提供标签推荐功能。

最后，我们考察了来自Stack Overflow网站的数据。我们讨论了几个这种论坛上实际面对的问题，这些问题只利用聚类算法就可以得到解决。

到这里为止，我们结束了Mahout中的聚类算法的介绍。我们循序渐进的介绍了机器学习的一个方面，一开始介绍基本的结构和数学知识，然后逐渐过渡到Hadoop集群上的大规模问题解决方案。

接下来我们将转向介绍Mahout中的分类，这需要更复杂的有关机器学习的理论和一些坚实的分布式和非分布式优化技术，这样才能完美的实现精确高效的分类。

Part 3 第三部分

分　类

第三部分也是本书最后一部分，由第13章到第17章组成，主要介绍如何利用Mahout进行分类。利用这一部分介绍的技术，你能够构建问题，并选择和准备合适的数据，以利用计算机自动将数据分到预定的类别中。分类是一种简单的决策形式，对单个问题提供一个离散的答案。基于机器的分类是上述决策的自动化过程，它从正确决策样本中进行学习并自动对决策进行仿真，这也是预测分析中的核心概念。分类依赖于有指导的学习，并且集中关注一次回答一个问题，这些特点使它有别于前面两部分分别介绍的聚类和推荐。与分类相比，聚类依赖于机器自身进行的决策，而推荐对多个可能的好答案进行选择和排序。

这一部分将介绍分类的3个阶段。第13章和第14章介绍分类的基本知识，并展示如何构建和训练Mahout分类模型。第15章介绍如何在整个过程中使用评估技术，来识别最好的模型版本和对性能进行调优。第16章和第17章介绍的是如何在真实场景下部署分类系统。这一部分介绍了如何正确识别和提取有用的数据，并详细探讨如何优化特征向量以便为多个Mahout分类算法所用。另外，这里还给出了一些避免诸如目标泄漏问题的提示。一步一步分析的示例可以让读者经历每一步过程，从而帮助其构建分类器和对性能进行调优。

除了详细解释如何为速度和规模需求很高的系统构建和部署高效、可靠的分类器，第17章还给出了一个实际的案例，在该案例中一个在线营销公司使用Mahout来完成需求。通过这个最后的案例，我们能够明白如何真正应用各章介绍的分类知识。

第13章 分 类

本章内容

- ❑ 为什么说Mahout具有强大的分类功能
- ❑ 分类的主要概念和术语
- ❑ 典型分类项目的工作流程
- ❑ 一个详尽的分类示例

生活常常给我们提出一些非开放式的问题，要求我们在特定选项中做出抉择。这种相对朴素的思想是人类和机器进行分类的基础。分类依赖于潜在答案的类别，基于机器的分类则是这种简单抉择的自动化形式。

本章介绍适合用Mahout做分类的情况，并解释Mahout相比于其他方法的优势。作为对分类的介绍，本章还解释了什么是分类，以及其中一些基本的术语和概念。此外，我们用一个实例来描述如何进行分类，介绍典型分类项目工作流程的三个阶段——训练、评估和调优模型——以及如何实地使用模型。然后，我们演示了一个简单的分类项目，并逐步展示如何把这些基本想法付诸实践。

13.1 为什么用 Mahout 做分类

Mahout广泛用于分类项目，但仅当训练样本个数非常大时，Mahout的优势才会体现出来。这里“大”的含义千差万别。对于100 000左右的样本，其他分类系统也能做到高效、准确。但是，当输入规模达到100万甚至1千万时，就需要像Mahout这种具有可扩展性的工具了。表13-1列出了一些最适合使用Mahout的场合。

表13-1 Mahout的最大价值在于可以处理其他方法无法处理的超大规模或增长迅速的数据集

系统中的样本规模	分类方法的选择
<100 000	传统的、非Mahout方法可以做得很好。Mahout训练速度可能更慢
100 000~1 000 000	可以考虑使用Mahout。灵活的API使Mahout成为一个理想的选择，尽管没有多少性能上的优势
1 000 000~10 000 000	在这个范围内，Mahout是一个极佳的选择
>10 000 000	其他方法完败于Mahout

Mahout在较大数据集上具有优势的原因在于，随着输入数据的增加，非可扩展系统用于训练的时间或内存需求并不是线性增长的。数据量达到两倍时，系统训练时间翻倍是可以接受的，但当5倍的输入数据导致100倍的训练时间时，就需要寻求其他的解决方案了。这种场合下，Mahout恰可以大显身手了。

一般情况下，Mahout的分类算法所需计算资源的增长速度不会超过训练或测试样本数。而且，大多数情况下所需的计算资源可以并行化。这样一来，你就可以在机器数量和运行时间之间找到平衡。

图13-1展示了像Mahout所提供的这类可扩展算法在大中型分类项目中的优势。当训练样本数很小时，传统的数据挖掘方法可以做得很好，甚至好过Mahout。但随着样本数的增加，Mahout可扩展与并行的算法会获得更好的时间效率。非可扩展算法所需的时间增长，通常是由于它们所需的内存会随着训练样本数无限制增长。目前，海量数据集变得越来越普遍。随着数据数字化存储技术的不断进步，数据采集的代价将大大减少。使用大量数据进行训练是件好事，因为通常可以提高准确度。结果，越来越多的大数据集需要采用可扩展的学习算法，而Mahout可扩展的分类器也正被越来越广泛的应用。

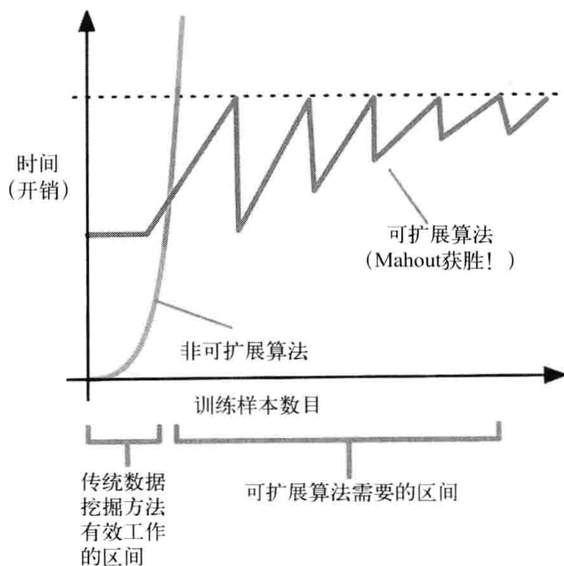


图13-1　在中大型分类系统中使用Mahout的优势。锯齿状曲线展示了增加机器带来的性能提升。每增加一台机器，都能减少训练时间

13.2　分类系统基础

在了解基于Mahout的分类方法之前，让我们更深入地了解一下分类的基本知识，以及它与推荐和聚类的区别。

首先，以我们熟悉的基于人工的分类为例。填写病历，完成一个企业服务质量的调查，或核对你在纳税申报表中是否勾选了正确的复选框，这都需要你从一组预先给定的答案中做出选择。有时可供选择的类别很少，甚至某些情况下仅仅需要回答“是”或“否”。在这种情况下，分类过程减少了潜在的庞大而复杂的可能性，以适应少量选项。

定义　分类是使用特定的信息（输入）从一个预定义的潜在回应列表中做出单一选择（输出）的过程。

比较一个品酒会上的人和一个去商店为晚餐买酒的人。在品酒会上，人们可能会问：“你觉得那个酒怎么样？”回答很可能多种多样：“辛辣”“水果味”“酿得不错”“刺激”“我姑姑沙发的颜色”或“品牌不错”。这个问题不具有特异性，而答案（输出）也不是从一个预先定义的列表中做出单一选择。这种情况就不是分类了，尽管它可能跟分类具有某些相似性。

现在考虑在杂货店买酒的顾客。在品酒会中探讨的一些酒的特性（口味、颜色等）可能对顾客也有意义，但为了高效地做出决定（买或者不买）然后回去就餐，顾客很可能参照心中一个简单的特性清单（红色、10美元、适合搭配比萨、一个熟悉的品牌等），然后对每一瓶酒做出决定（买或者不买）。他们可能会忽略一些细节和细微差别，但对于可选答案极少的情况（买或不买），对问题进行简化是一个切实可行的办法。在这种情况下，顾客可以及时离开商店回家吃饭。这就是一种分类。

机器不能完成人类所有的思维和决策。但是，在特定条件下，机器可以模仿人的决策，高效地进行分类。当需要在有限范围内做出单一选择时，这种基于机器的方法是可行的。输入可以看做是一组特定的特性，输出则是明确的，是简短列表中的单一选择。

分类算法是预测分析（predictive analytic）的核心。预测分析的目标就是建立一个自动化系统，以取代人类做出决策的功能。分类算法是实现这一目标的一个基本工具。

垃圾邮件检测就是预测分析的一个例子。计算机使用用户历史细节和电子邮件内容特征来做判断：一封新电子邮件是垃圾邮件，还是比较受欢迎的邮件？另一个例子是信用卡欺诈检测。计算机用账户最近的历史和当前交易细节来确定是否是欺诈性交易。

自动化系统如何做到这点？我们希望它们神奇地做到这一点，但实际上依赖于让它们从样本中学习。

定义　计算机分类系统是一种机器学习的形式，它通过学习算法使计算机基于经验做出决策，这一过程模仿了人的决策。

分类算法基于样例学习，但它们并不能取代人的判断，这在很大程度上是因为它们需要精心准备一批正确决策的样本，并从中学习。而且，它们也需要精心准备用于判断的输入。与Mahout的其他用途不同，基于机器的分类是一种有监督的学习形式。

13.2.1 分类、推荐和聚类的区别

分类与前面各章中提到的聚类是对立的，因为聚类算法自己能决定哪些区别是重要的（在机器学习的范畴中，它们是无监督学习算法），而分类算法则需要模仿做出正确决策的样本来学习（它们是有监督算法的）。分类算法与推荐算法也是有区别的，分类算法试图从很有限的输出集合中做出单一决策，而推荐算法会选择许多可能的答案，并对其进行排序。

警告 理解分类概念的一种方式是提醒自己分类**不是**什么：如果你的系统是无监督的，那么它就不是分类；如果问题是开放式的，或者答案没有类别，那么你的系统也**不是**分类。

有点奇怪的是，分类可以作为一种创造性的、有效的工具用于推荐过程。通常情况下，建立一个基于分类的推荐系统并不如真正的推荐系统效率高，但用作推荐程序的分类器可以使用常规推荐系统很难使用的某些信息。推荐系统通常使用用户行为历史，而无视用户和物品的特点，而后者对于分类系统却是很有用的。第17章中的案例研究描述了一个分类器如何有效地用作推荐程序。

我们已经回答了“什么是分类”这个问题。接下来，我们了解分类用来干什么。

13.2.2 分类的应用

预先定义一个小的类别集合，分类系统的输出就是将其输入数据对应到此集合中的一个类别上。分类的效用通常由预定义类别的意义来衡量。建立了数据与类别的对应关系之后，计算机或用户可以在此输出的基础上做进一步的处理。

分类通常用于预测或检测。以信用卡诈骗为例，分类器能基于已知的购买行为或其他信用卡交易样本，有效地预测交易是否属于欺诈。特定交易是否属于欺诈行为，可以表示为两个类别：是，欺诈；否，没有欺诈。

在某些行业中，如保险或电信，分类对于预测公司的客户流失很有用。我们可以通过前几年的客户历史数据为这类应用训练分类系统，其中客户的“流失”/“保持”对应于目标类别集合。训练数据集中，能够准确预测用户是否会继续使用服务的客户特性被选作特征。一旦训练完成，分类系统可基于特定的已知特性，包括当前行为，对其他用户未来的行为进行预测。

注意 前面我们用预测这个词来描述分类器的行为。它除了指代预见未来事件的能力，也可以指对一个可能存在，但在分类时并不容易获得的值的估计。

我们可以将预测看做这样一个任务，它基于现有数据来为一个特性赋值（分配一个类别），而不是直接测定该特性的值。通常直接对该特性本身进行判别的代价很高，这也是用分类做预测的原因。这里，高代价指金钱或时间成本太高，或非常危险。

例如，对于糖尿病等会逐渐恶化的疾病，你可能不希望花一年或更长时间去观察其恶化的过程。相反，你可以建立一个分类系统，尽早评估将来可能出现的风险，而不必等到疾病恶化到不可挽回的地步。知道病人是糖尿病早期或可能发展为糖尿病，医生就可以采取预防措施或为病人进行早期的治疗。

提示 分类通常是一种通过分析低代价变量组合来确定高代价变量值的方法。

在某些场合，分类系统能够降低对人身的伤害。例如，通过认知测试或PET扫描这类无创手段诊断老年痴呆症，显然要比直接解剖大脑的手段更可取，尤其是在病人还活着的时候。

因此，分类是很有用的。但是，为什么要用Mahout来做分类，而不用其他软件呢？

13.3 分类的工作原理

构建分类系统主要有两个阶段：通过学习算法建立一个模型，然后使用该模型对新数据进行分类。如何选择训练数据、输出类别（目标）、系统使用的学习算法以及用作输入的变量，是构建分类系统第一阶段的关键。

构建分类系统的基本步骤如图13-2所示。该图展示了分类过程的两个阶段：上面表示训练分类模型的过程；下面的过程为该模型提供新的输入样本，然后通过为样本分配类别（目标变量）来模仿决策。训练算法的输入由样本数据和标注的已知目标变量组成。当然，在生产环境中新样本的目标变量是未知的。在对算法进行评估时，这些目标变量的值是已知的，但对分类模型不可见。

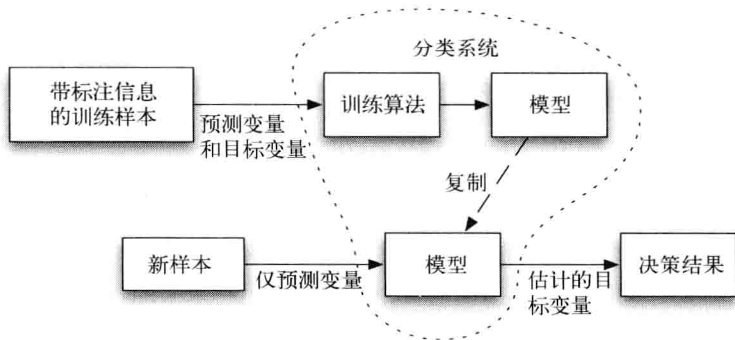


图13-2 分类系统的运转过程示意图。虚线框内的部分是分类系统的核心——训练算法训练一个模型以模仿人类的决策。该模型的一个副本将被用于评估或生产环境下，对于新的输入样本，它能对目标变量进行估计

测试时提交给模型的新样本不能包含在训练数据中，这一点没有显式地在图13-2中表示出来。通过比较模型选择的结果与已知的目标变量值，就可以评估模型的性能，这一过程会在第15章进行深入探讨。

不同人描述分类的术语大不相同。为保持一致性，我们限定了本书中用于表述关键概念的术语，其中大部分术语都列于表13-2中。特别需要注意记录和字段的关系：记录存储的是与训练样本或生产样本有关的所有值，字段则存储每个样本的某个特征值。我们将在接下来的几节中对表13-2中列出的要点进行探讨。

表13-2 分类中的关键概念术语

关键概念	描 述
模型	一个做决策的计算机程序；在分类中，训练算法的输出就是一个模型
训练数据	训练样本的一个子集，带有目标变量值的标注，用作学习算法的输入以生成模型
测试数据	留存的部分训练样本，隐藏其目标变量值，以便用于评估模型
训练	使用训练数据生成模型的学习过程。随后该模型可将预测变量作为输入来估计目标变量的值
训练样本	具有特征的实体，将被用作学习算法的输入
特征	训练样本或新样本的一个已知特性。一个特征与一个特性是等同的
变量	在这个上下文中，指一个特征的值或一个关于多个特征的函数。这一用法不同于计算机编程中的变量
记录	存储一个样本的容器。这样一个容器由多个字段组成
字段	记录的一部分，包含一个特征的值（变量）
预测变量	选做分类模型输入的一个特征。不是所有的特征都会被用到。某些特征可能是其他特征按某种规则进行组合的结果
目标变量	分类模型试图去预测的一个特征：目标变量是可分类的，决定其类别就是分类系统的目标

在接下来的几节中，我们更密切地关注模型是什么，以及它是如何训练、测试并用于生产环境的。你将学会区分预测变量和目标变量，理解如何正确地使用记录、字段和值，并了解变量可取值的4种形式。另外，你会发现Mahout分类恰是监督学习的一个示例。

13.3.1 模型

分类算法从样本中进行学习的过程就是训练。这一训练过程的输出称为模型。模型是一个函数，随后可作用于新样本以生成输出，模仿原始样本上的决策。这些模仿的决策就是分类系统的最终产出。

注意 训练算法生成的模型本身就是一个有效的计算机程序。

图13-2已经展示了如何将训练样本和标注的输出值传递给分类系统的学习部分以生成模型。这个模型本身就可以看做是做决策的计算机程序，它会被复制并接受新样本（作为输入）。我们的目的是让模型模仿训练样本作为输入的例子对新样本进行分类。

13.3.2 训练、测试与生产

实际上，训练样本通常分为两部分。其中一部分用作训练数据，大概占可用数据的80%～90%。

训练数据用于训练以产生模型。另一部分称为*测试数据*，随后会被提交给模型，但不告诉模型真实的答案——尽管它们是已知的。这么做是为了比较模型的输出和期望的输出。

一旦模型的性能满足需求，它就会被投入到生产中，对其他样本进行分类，而这些样本的正确决策值是未知的。生产环境中的结果通常不会100%准确，但输出的质量应该与测试阶段的结果保持一致，除非输入数据的质量或其他外部条件发生了改变。

一个常用的做法是随着时间的推移，将采样生产环境中的样本并将其加入到训练数据中，这样可以生成不断更新的模型版本。当外部条件可能发生改变并导致决策质量随时间推移不断下降时，这种对生产样本进行采样的做法十分重要。在这种情况下，我们可以训练一个新的模型，使决策质量得到改善或恢复到原来的水平。

13.3.3　预测变量与目标变量

*变量*是一个样本某个特征或特性的值。这个值可以是通过测量或计算得到的。分类器的最终目标是估计一个特定问题的分类答案，这个答案就是*目标变量*。在生产环境中，目标变量是需要为每个新样本寻找的值。在训练阶段，每个历史数据（训练数据）的目标变量都是已知的，并被用来训练模型。

在分类中，*预测变量*是为模型提供的线索，以便模型能判断为各个样本赋一个什么样的目标变量。用于分类的预测变量也称为*输入变量*（input variable）或*预测因子*（predictor）。

用来分类的样本特性也称为*特征*；对一个特定特性进行描述，以便用于分类系统的过程叫做*特征提取*。如果一个特征被选作模型的输入，那么我们可以将该特征的值看做预测变量。

回忆前面那个顾客为晚餐买酒的场景。酒的属性（如“颜色”、“适合搭配牛排”、“适合搭配鱼”、“适合搭配比萨”），以及其他属性（如价格），都被顾客用于做决策（买或不买）。在购买决策过程中，酒的这些特征类似于机器分类中模型输入的预测变量。类比到这个例子，决策有一个二值目标变量：买或者不买。每瓶酒仅对应一个选择。

注意　*在训练时目标变量和预测变量都会提交给学习算法。然而，在测试和生产过程中，仅有预测变量是对模型可见的。*

在准备分类系统时，大多数用作训练数据的历史样本都会有一个目标变量值的标注。在测试过程中，测试数据的目标变量会被保护起来。对比模型估计的目标值与各测试样本已知的目标值，可以反映分类模型的精度。

对于分类，表示目标变量的字段必须有一个类别型（categorical，也常常译为指称型）的值。而预测变量的值则可以是连续的，或者类别型，或者文本，或者单词。你会在13.3.5节了解这些值的类型（连续、类别型、文本型或单词型）。通常，目标变量是一个二值类别变量，也就是说，它仅有两种可能的取值。垃圾邮件检测就是一个有着二值类别型变量的分类系统的例子：邮件是垃圾或不是垃圾。当目标变量有两个以上的可能值时，称为多类分类问题。第14章（14.4节和14.6节）有一个使用取自20 Newsgroups数据的多类分类的例子。

13.3.4 记录、字段和值

构成分类算法输入的样本通常表示为记录的形式。我们可以将记录看做存放一个样本值的仓库（无论是训练样本，还是生产环境中使用的新样本）。记录通常包含一些命名字段，每个字段都有一个值。

每个训练样本的记录都会有一些字段，用于存放目标变量和一个或多个预测变量，如图13-3所示。当然，一个生产样本中目标变量的字段是未知值，需要由分类算法来确定。生产过程中模型的输出就是目标变量的估计值，在图中用T表示。每个输入样本都会相应地有一个输出。

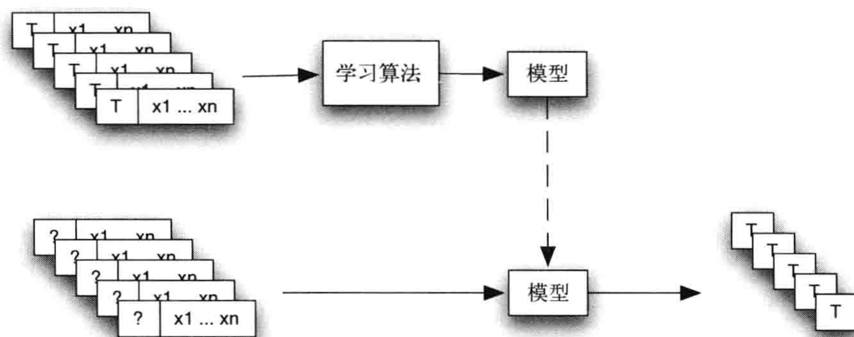


图13-3 分类模型的输入与输出：目标变量（T）与预测变量（x1 … x*n*）。注意，在训练阶段T是包含在输入数据中的，而生产阶段则从输入数据中剥离（用?表示）

尽管目标值只能是类别型，但表示用作预测变量的特征则可以是多种不同类型的值。下面我们会对它们进行比较。

13.3.5 预测变量值的 4 种类型

预测变量在学习算法中用作输入，但哪个变量有用则部分取决于值的类型。区分变量值的类型不仅仅是一个学术上的问题。正确识别值的类型有助于改善分类系统。

预测变量的值有4种常见类型，即连续型、类别型、单词型和文本型，参见表13-3。

表13-3 用来表示特征的4种常见类型

值的类型	描 述
连续型	这是一个浮点值。这类值可能是价格、重量、时间，或其他任何具有数值大小的量，且这个大小是值的关键属性
类别型	一个类别型值可以从一个预先指定的集合中取值。尽管类别型值的集合可以非常大，但通常这个集合很小，而且可能就只有两个元素。布尔值通常被视为类别型值。另一个例子是供应商ID
单词型	单词型值就像类别型值，但它可取值的集合是无限的
文本型	文本型值是多个单词型值的序列，全都是同一类型。文本是文本型值的典型示例，但一个电子邮件地址列表或URL列表也是文本型的

区分连续型值和类别型值通常比较困难。通常，数字识别码会被误认为连续值，但实际上它们应该被视为类别型值。举个例子，考虑下面这些值，它们表示邮政编码的前三个数字：757、415、215、809以及446。这些值看起来都是数值型值，你倾向于把它们视为连续的，但它们更符合类别型的描述，因为每个值都取自一个预先指定的值的集合。将一个类别型值视作连续型值会严重影响分类器的精度，反之亦然。

提示　像整数这样的对象未必是连续的。一个重要的检验标准是将两个值相加，或取一个值的对数或平方根。如果得到的结果是未定义的，那这很可能是类别型值，而非连续型值。另一个提示是，任何有相关计量单位的度量通常都是连续变量。

单词型值是类别型值的扩展，这种情况下有许多可能的值。二者之间的区别在于一个特征可取的值有多少个。即使某个特征可以取很多个值，就会存在有限的预先指定值表吗？当你无法确定有多少个可取的值时，变量就很可能是单词型的。通常，我们很难判定应该将一个值视为单词型还是类别型。例如，汽车的品牌或型号似乎是类别型的，但新的汽车品牌或型号随时可能出现，因此把这样一个值视为单词型可能是更好的选择。

在某些场合，类别型可取的值或许有几千个，甚至更多，通常把它们视为单词型更合适。如果你已经有许多可能的值，很可能值的集合并不像你想象的那样固定。有时候，类别型值可能有某种顺序，但Mahout中的分类算法目前并没有考虑这种顺序。

如果有这样一个变量，其值可以被视为单词型值的集合，你应该考虑将它看成是文本型，尽管仅在少数样本中会出现一个以上的这种值。文本型值通常存储为字符串，因此需要词条化工具将字符串转换成单词型值的序列。

提示　通常可以基于这样一个事实来判定变量是文本型的，即值的成分可以以任意组合的形式出现。基于这一点，一个可取值为General Motors、Ford或Chrysler的值并非文本型，尽管General Motors包含了两个词。这是因为像General Chrysler Ford这样的值无效。

查看表13-4时请回顾一下表13-3中的描述，前者列出了取自不同数据源的一些样本。第一个值是一个电子邮件地址，是取自开放式集合的单个实体，因此是单词型。垃圾邮件单词值或非垃圾邮件单词值由一个单词列表（或可能的单词）组成，因此被认为是文本型的。

表13-4　4种类型值的样本数据。这些样本是典型的电子邮件数据的特征

名　称	类　型	值
from-address	单词型	`George`
in-address-book?	类别型（`TRUE`、`FALSE`）	`TRUE`
non-spam-words	文本型	`Ted, Mahout, User, lunch`
spam-words	文本型	`available`
unknown-words	连续型	`0`
message-length	连续型	`31`

注意 关于一个特定算法中会使用何种类型的预测变量，详见14.5节的讲解。

学习是有监督的还是无监督的，是分类和其他Mahout功能（如聚类或推荐）的一个区别。下一节会解释这个术语。

13.3.6 有监督学习与无监督学习

分类算法与聚类算法（如前面各章中提到的k-means算法）相关，但又大不相同。分类算法是一种有监督学习，这与聚类算法的无监督学习正好相反。有监督学习算法处理的是带有期望目标变量值的训练样本。无监督算法则没有期望的答案，取而代之的是对数据的合理解释。

有监督学习算法和无监督学习算法通常可以有效地结合起来。聚类算法可以用来生成为学习算法所用的特征，或者多个分类器的输出可以作为特征为聚类算法所用。此外，聚类系统通常会建立一个可用于对新数据进行分类的模型。该聚类系统模型的工作方式与分类系统生成的模型非常类似。不同之处在于生成模型的数据。对于分类，训练数据包含了目标变量。而对于聚类来说，训练数据不包含目标变量。

现在你应该对分类是什么有了一个基本的认识，并且知道使用Mahout对超大规模数据集进行分类的优势；是时候把这些想法付诸实践了。

13.4 典型分类项目的工作流

虽然具体项目之间会存在差异，但构建并运行分类系统都需要遵循一系列非常标准的步骤。在这一节，我们给出了分类项目的典型工作流：训练模型，评估并调整模型以达到可接受水平，然后在生产环境中运行系统。我们将使用合成数据阐述工作流程中的一些步骤。最后，13.5节给出了一个更完整的示例，用Mahout完成分类项目。

作为任何项目的先决条件，你需要确定要寻找的信息（目标变量）是否能够当做特征，以适当形式存储在一条记录中，并且符合总体目标的要求，比如识别欺诈性的金融交易。有时候，你需要根据一些现实因素调整目标变量的选择，例如训练代价、隐私问题等。你也需要知道有哪些数据可用于训练，系统运行时又会遇到什么新数据，这样才能设计出有用的分类器。

注意 通常，构建分类器的大部分精力会花在设计并提取有用的特征上。这一步所需的工作量也常常超出预期。

分类项目的一般开发步骤如表13-5所示。系统每个阶段的准确性和输出结果的有效性，很大程度上反映了用作预测变量的原始特征选择和目标变量的类别选择的好坏。对于每种选择而言并不存在单一的正确选择：有很多可取的方法和调整手段，要构建一个高效、健壮的系统，需要尝试一些不同的方法。

表13-5　典型分类项目的工作流

阶　段	步　骤
(1) 训练模型	定义目标变量 搜集历史数据 定义预测变量 选择学习算法 使用学习算法训练模型
(2) 评估模型	在测试数据上运行 调整输入（改变预测变量、改变算法，或二者一起改变）
(3) 在生产中使用模型	输入新样本，估计未知的目标变量值 在必要时重新训练模型

训练和评估模型的过程中，每一步的特征选择都需要综合考虑经济和时间上是否负担得起，以及模型估计值的精度是否可以接受。

在下面的几节中，我们将会具体讨论分类的3个阶段。

13.4.1　第一阶段工作流：训练分类模型


No. 12

在分类项目的第一阶段，你需要心中有一个目标变量，这有利于选择合适的历史数据用于训练，以及选择有效的学习算法。这些决策之间都是密切相关的。

在下面的讨论中，我们将考察特征选择方法对Mahout学习算法的具体影响方式，确定哪个Mahout学习算法适合当前的分类器。

1. 定义目标变量的类别

定义目标变量涉及目标变量类别的定义。目标变量不能在一个开放集合中取值。你选择的类别，反过来也会影响你对学习算法的选择，因为某些算法仅适用于二值目标变量的情况。目标变量的类别数越少越好，如果可以把类别数降到两个的话，那么会有更多的学习算法可供选择。

如果目标变量不适合降低到简单形式，你可能需要构建多个分类系统，各自处理你预期目标变量的某个方面。在你最终的系统中，你可以把这些分类系统的输出组合起来，以提供所需的全面而细致的决策或预测。

2. 搜集历史数据

你所选择的历史数据源，部分依赖于搜集带标注历史数据的具体需求。在某些问题中，确定目标变量的值是很困难的。

俗话说垃圾堆里出垃圾，放在这里很合适，你需要小心谨慎地确保历史数据中目标变量值的准确性。在一个有缺陷的目标变量或数据搜集过程基础上建立了很多模型之后，由于模型的效果很差，你可能并不想再构建其他模型了。

3. 定义预测变量

选择了一个有效的目标变量并定义好其可选值集合之后，你需要定义预测变量。这些变量是从训练和测试样本中提取的特征的具体编码。

你需要再一次检测样本源，确保它们的特征有效，且它们的值能以适当的格式存放在记录中。图13-3给出了用于训练、测试数据以及生产数据的记录中的预测变量。

警告 选择并定义预测变量的一个常见错误是在预测变量中包含**目标泄漏**（target leak）。

定义预测变量的一个重要考虑因素是避免导致目标泄漏。目标泄漏，顾名思义，是一个错误，指在选择预测变量时，无意中引入了目标变量的信息。此错误与在训练样本中有意包含目标变量不同。

目标泄漏会严重影响分类系统的精度。当你告诉分类器目标变量是一个预测变量时，这个问题会相当明显。这是一个显而易见的问题，但它经常发生。

目标泄漏可能很微妙。假设需要构建一个垃圾邮件检测器，你将垃圾邮件和非垃圾邮件样本放入不同的文件，随后在各个文件中按顺序给样本标上序号。同一文件中的样本将会有着某个范围内的连续序号，而这一点可以用来区分垃圾和非垃圾邮件。如果存在这种类型的目标泄漏，很多分类算法都会迅速发现序号的范围与垃圾/非垃圾邮件的对应关系，并仅依赖该特征解决问题，因为它（在训练数据中）看起来是完全可靠的。这就是问题所在，当你将新数据传递给一个几乎完全依赖序号的模型时，这个模型只能举手投降了，因为新样本中的序号可能不在前面的范围之内。该模型很可能将新样本归入具有最大序号的范围。

目标泄漏可能是相当隐晦的，很难找到。最好的建议是，对结果太理想的模型持怀疑态度。

4. 例1：将位置用作预测变量

我们用一个使用合成数据的简单例子演示如何选择预测变量，以使Mahout习得的模型能够准确地预测期望的目标变量。图13-4中是一个历史数据集合。假设你在搜索颜色填充的形状——颜色填充是目标变量。这个问题中，什么特征最适合用作预测变量呢？

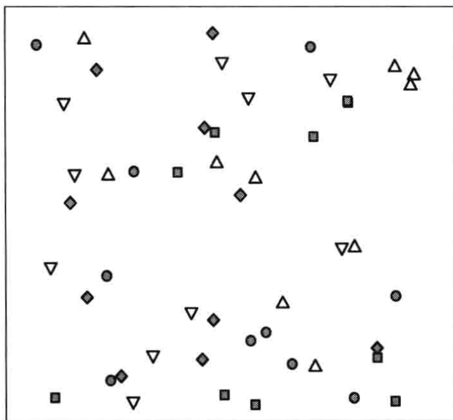


图13-4 使用位置来对颜色填充进行分类：在这个历史数据中，目标变量是颜色填充，特征可以视为包含形状和位置的预测变量。位置看起来更适合用作预测变量——水平（x）坐标可能就足够了。形状似乎并不重要

对于分类问题，你必须确定目标变量的类别，在这个例子里就是颜色填充。颜色填充很明显有两种可能取值，即填充或未填充。你也可以用一个答案为是或否的问题作为目标变量："元素是否填充颜色？"

不要认为这一点是理所当然的，因此请检查历史数据，确保它包含了目标变量。这里它明显包含了（很好！），但现实世界中并不总是如此明显。

现在你需要选择用作预测变量的特征。哪些特征是你可以正确表述的呢？首先排除颜色填充（它是目标变量），你可以将位置或形状用作变量。你可以用*x*和*y*坐标来描述位置。基于一个数据表，你可以为每个样本创建一条记录，包含目标变量和你正在考虑的预测变量的字段。图13-5是两个训练样本的记录示例。（这个例子的完整数据可以在Mahout源代码的examples模块中找到。）

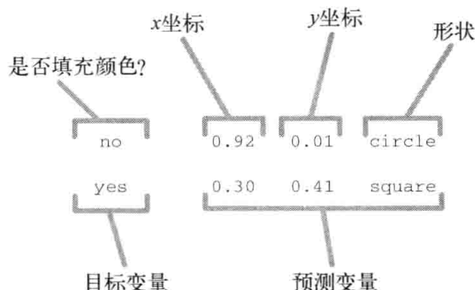


图13-5　训练数据中的两条记录。图13-4中数据的记录包含存储目标变量值的字段，以及存储预测变量值的字段。在这种情况下，与位置（*x*、*y*坐标）和形状相关的值包含在两个样本中

如果再次查看图13-4所示的历史数据，你会发现尽管训练数据同时具有位置和形状信息，*x*坐标就足以用于区分填充和未填充的符号了。形状对于判定一个符号是否被颜色填充不起作用，*y*坐标也是如此。

提示　不是所有特征都对每一个分类问题有用。特定场合和你的具体问题决定了哪些特征会是有效的预测变量。了解你的数据是成功的关键。

在设计分类系统时，你需要根据经验选择最可能有效的特征，模型的准确性也会反映出你的选择是否正确。没有必要排除所有没有区分力的特征，但是引入的冗余或无关特征越少，分类器提供准确结果的可能性就越大。例如图13-4中的数据，可能最好忽略*y*坐标和形状，仅使用*x*坐标训练模型。

5. 例2：不同的数据需要不同的预测变量

仅仅因为新搜集的数据跟以前的数据具有同样的特征，就使用跟以前同样特征的预测变量是不对的。这一观点在图13-6中得到了体现。这里你可以看到另一组历史数据，它们与之前的数据

有着同样的特征。但在这种情况下，无论x还是y坐标似乎都对预测符号是否填充颜色没有作用。位置不再有用，但现在形状成为了一个有用的特征。

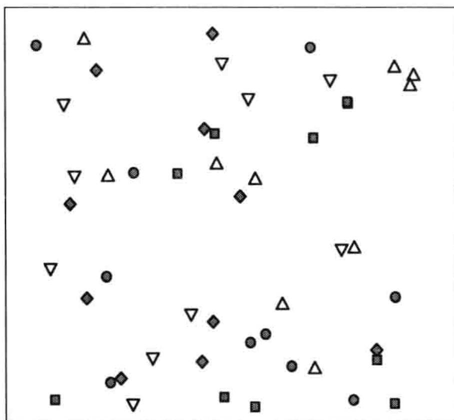


图13-6 使用形状对颜色填充进行分类：这个数据与图13-4中的数据使用同样的特征（形状和位置），但预测目标变量（颜色填充）需要使用不同的预测变量。这里，位置不再有效，但形状很有用

在这个例子中，选作预测变量的特征（形状）具有3种值（圆形、三角形、正方形）。如果有必要，你可以进一步引入朝向，来区分这些形状（方形与菱形；朝上的三角形和朝下的三角形）。

6. 选择一个学习算法来训练模型

在任何项目中，你都必须在选择所要使用的算法时考虑一些参数，例如训练数据的规模、预测变量的特性，以及目标变量的类别数等。Mahout的分类算法包括朴素贝叶斯（naive Bayes）、补充朴素贝叶斯（complementary naive Bayes）、随机梯度下降（Stochastic Gradient Descent，SGD）以及随机森林（random forest）。其中朴素贝叶斯、补充朴素贝叶斯以及SGD的使用和效果会在14.5节详细讨论。

即使是前面那些简单的例子，不同的算法也各有优势。在例1中，训练算法应该使用x坐标位置来判定颜色填充。在例2中，形状更有用。然而，一个点的x坐标点位置是连续变量，某些算法可以很好地使用连续变量。在Mahout中，SGD和随机森林法就是如此。另一些算法则无法使用连续变量，例如朴素贝叶斯和补充朴素贝叶斯。

当数据集规模特别大时，并行算法对速度的提升很明显。那么，为什么有时也会用非并行（串行）算法做分类呢？答案很简单，因为有些东西需要我们做出权衡。并行算法有相当大的额外开销，就好像它开始处理样本之前，需要花一些时间去“找到车钥匙，然后下到车道”。即使你可以接受这个时间上的开销，对于某些中等规模的数据集，串行算法可能不仅仅是够用，而往往是首选的。这种权衡如图13-7所示，其中比较了假设的串行和并行可扩展算法的运行时间。

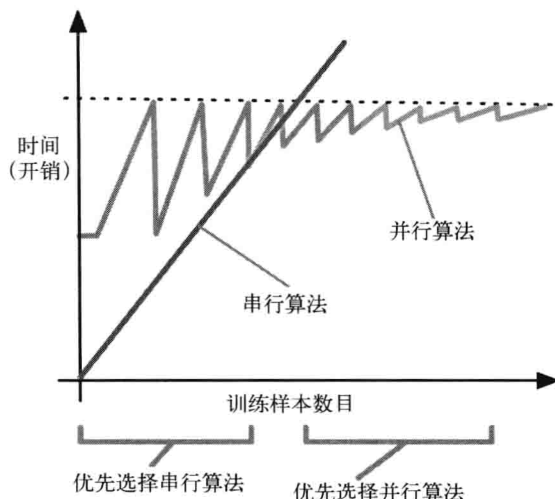


图13-7　对比两个Mahout分类算法，两种算法都是可扩展的

在项目规模较小时，串行算法在时间上比较有优势。在中等规模时，两种算法性能都比较理想。当训练样本数变得非常大时，并行算法要优于串行算法。

我们在图13-1中解释过，并行算法时间开销曲线中锯齿形状的下降部分是由于添加了新机器。

7. 使用学习算法训练模型

确定并封装好合适的目标变量和预测变量集，并选好了学习算法之后，训练的下一步就是运行训练算法来生成模型（这些步骤参见图13-3）。这个模型会捕获学习算法所能看清的预测变量与目标变量之间关系的本质。

对于例1，模型可能是类似下面的伪代码：

```
if (x > 0.5) return FILLED;
else return UNFILLED;
```

当然，学习算法所产生的真实模型并不会以这种方式实现，但这展示了本例中可以工作的规则有多简单。

13.4.2　第二阶段工作流：评估分类模型

在生产中使用分类系统之前有一个必要步骤，即确定它到底能有多好的表现。要做到这一点，你必须评估该模型的准确性，并在真正开始分类之前做出或大或小的调整。

评估训练好的模型通常并不简单。第16章详细介绍了在准备将系统投入生产之前，如何对模型进行评估和调优。13.5节有一个循序渐进的例子，你将初步了解如何评估。

13.4.3　第三阶段工作流：在生产中使用模型

一旦模型输出的准确性达到可接受的范围，我们就可以对新数据分类了。生产中分类系统

的性能取决于很多因素，最重要的一个因素是输入数据的质量。如果需要分析的新数据预测变量的值不准确，或者新数据与训练数据不一致，或外部条件随着时间发生了改变，分类模型输出的质量都会下降。为避免这些问题，对模型进行周期性的复检是很有用的，有时有必要重新训练模型。

要进行复检，你需要搜集新的样本，并验证或生成目标变量值。然后可以像第二阶段一样，将这些新样本作为测试数据，比较这些结果和原始数据中保留用于测试的数据集的结果。如果新的结果明显恶化，这就意味着某些因素发生了变化，你需要考虑重新训练模型。

注意需要重新训练模型的一个常见原因：将模型集成到你的系统时会极大幅度地改变系统。举个例子，如果一家大银行部署了一个非常有效的欺诈检测系统，会怎样？在数月或数年之内，骗子会慢慢适应并开始采用原始模型无法检测的新技术。在这些新的欺诈方法造成重大损失之前，银行的建模人员监测到精度下降并更新模型是很重要的。

一个分类系统的性能有所降低时，并不一定就要放弃这种方法。使用新的、更合适的训练数据重新训练模型也许就足够了。另外，对训练算法做出某些调整可能会有用。实时评估也是重新训练的一部分，16.4.1节讨论模型更新时会有更具体的阐述。

13.5 循序渐进的简单分类示例

现在你已经掌握了一些基本知识，包括分类是什么，以及典型项目的工作流程。你也看到了针对要处理的问题，仔细选择用作预测变量的特征和选择待估计目标变量的重要性。

虽然Mahout的目标是处理海量数据集，但先从一个简单的例子入手，更有助于熟悉Mahout。这一节将带你在小规模真实数据集上用Mahout实现一个分类器。下面我们会介绍使用的数据、如何训练模型，以及如何调优分类器以使其更加准确。

13.5.1 数据和挑战

这一节带你训练一个分类模型，来解决填充颜色的判定问题；这里使用与前面章节类似的合成数据。在训练数据中，目标变量仍然是填充颜色，有两个类别：*填充*和*未填充*。在这个数据集中，位置是预测颜色填充的关键，我们将使用Mahout中的随机梯度下降（SGD）分类算法来训练模型。

这个DIY实践项目中的历史数据如图13-8所示，是Mahout发行版的一部分。为帮助你学会使用分类，Mahout在包含示例程序的JAR文件中集成了几个这样的数据集。我们将这类数据集称为*甜面圈数据*（donut data）。

在这个简单的分类项目中，目标变量是填充颜色，你的任务是构建一个可以找到填充点的系统。

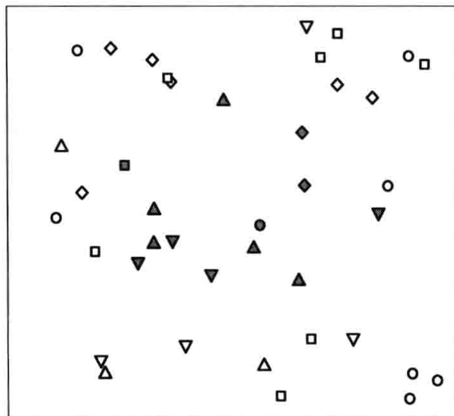


图13-8　例3，用于简单分类项目的数据：甜面圈数据。请思考一下，哪些特征对于寻找填充点（即实心点）有用呢

13.5.2　训练一个模型来寻找颜色填充：初步设想

在思考如何为项目训练模型时，请想想什么样的预测变量会有效。在13.3.1节讨论并示于图13-4的例子中，关键是位置，而非形状。粗略地看下图13-8你就会知道，在这个简单的数据集中位置仍然是关键。但是你可能会有些疑惑，在这个例子中，无论是*x*坐标还是*y*坐标，还是二者的组合，都不足以预测填充点的位置。

对于像位置这样的特征，用作变量的方式不只一种。也许把位置定义为到特定点——如A、B或C——的距离会比较有效，如图13-9所示。

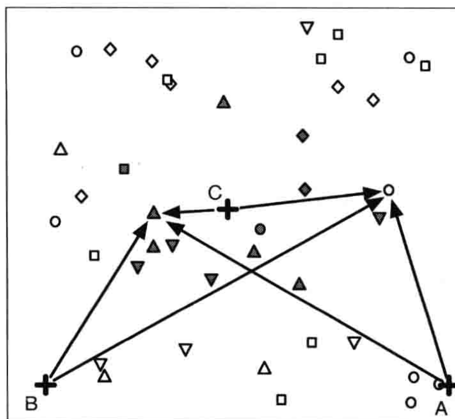


图13-9　如果用位置作为这个甜面圈数据集的特征，你必须根据特定目的——确定填充点——选定最合适的位置。在本例中，到点C的距离看起来最有效

尽管x和y坐标足以定位任何一个点，但需要预测一个点是否填充时，用它们做特征并不合适。引入到点A、B和C的距离重新表示位置，更易于为这个数据集构建准确的分类器。有到点C的距离就差不多了，但你也可以在系统中加入额外的预测变量。具体做法将在接下来探讨。

13.5.3 选择一个学习算法来训练模型

Mahout提供了大量的分类算法，但很多都是为处理海量数据而设计的，因此用起来有点麻烦——至少在起步阶段是这样的。不过，也存在一些易于上手的算法，尽管仍然保持了可扩展性，但它们在小数据集上的额外开销会比较少。

用于Logistic回归的随机梯度下降（SGD）算法就是这样一个低开销的方法。它是一个串行（非并行）算法，但是正如图13-9所示，它很快。SGD支持大数据处理的很重要一点，是它仅需要固定量的内存，而不受输入规模的影响。

1. 开始运行Mahout

首先，从mahout.apache.org下载Mahout并解压。假设你已经设定环境变量MAHOUT_HOME的值为解压后的目录路径，那么可以通过Mahout的命令行工具列出可用的命令：

```
$ $MAHOUT_HOME/bin/mahout
An example program must be given as the first argument.
Valid program names are:
  canopy: : Canopy clustering
  cat : Print a file or resource as the logistic regression models would see
     it
...
  runlogistic : Run a logistic regression model against CSV data
  ...
 trainlogistic : Train a logistic regression using stochastic gradient
     descent
 ...
```

这里我们最为感兴趣的命令包括cat、trainlogistic和runlogistic。

2. 查看Mahout的内建数据

为帮助你着手进行分类，Mahout将一些数据集作为资源纳入到包含示例程序的JAR文件中（参见examples/src/main/resource/目录）。当你为SGD算法指定一个输入时，如果该输入没有对应到一个现有的文件，但该名称存在于资源中，训练或测试算法会改从资源中读取该名称所对应的数据集。

要列出其中任何资源的内容，可以使用cat命令。例如，下面的命令会将图13-8所示甜面圈数据集的内容列出来：

```
$ bin/mahout cat donut.csv
"x","y","shape","color","k","k0","xx","xy","yy","a","b","c","bias"
0.923307513352484,0.0135197141207755,21,2,4,8,0.852496764213146,...,1
0.711011884035543,0.909141522599384,22,2,3,9,0.505537899239772,...,1
...
0.67132937326096,0.571220482233912,23,1,5,2,0.450683127402953,...,1
0.548616112209857,0.405350996181369,24,1,5,3,0.300979638576258,...,1
0.677980388281867,0.993355110753328,25,2,3,9,0.459657406894831,...,1
$
```

注意，这里大部分内容都没有显示出来，而是用省略号（…）代替了。这个文件的第一行指定了数据各个字段的名称，随后各行就是数据本身了。如你所见，我们提到的原始预测变量包括：x、y、shape（形状）和color（颜色）。*x*和*y*是范围[0,1]内的数值，而形状则是范围[21,25]内的整数。颜色是一个整数，且只能是1或2。因为用一个正方形减去一个三角形没有意义，所以我们将形状变量看做是可分类的。同样，颜色值也应该被视为仅有两个可能取值的类别变量的数值编码。

除了x、y、shape和color，这个数据集中还有其他变量，以便你在实验中尝试不同的分类方案。这些变量的描述详见表13-6。

表13-6　donut.csv数据文件中的字段

变　量	描　述	可 能 值
x	一个点的*x*坐标	从0到1的数值
y	一个点的*y*坐标	从0到1的数值
shape	一个点的形状	从21到25的形状代码
color	点是否被填充	1表示“空”，2表示“填充”
K	仅用*x*和*y*进行k-means聚类所得到的ID	从1到10的整型簇ID
k0	用*x*、*y*和color进行k-means聚类所得到的ID	从1到10的整型簇ID
xx	*x*坐标的平方	从0到1的数值
xy	*x*和*y*坐标的积	从0到1的数值
yy	*y*坐标的平方	从0到1的数值
a	到原点(0,0)的距离	从0到 $\sqrt{2}$ 的数值
b	到点(1,0)的距离	从0到 $\sqrt{2}$ 的数值
c	到点(0.5,0.5)的距离	从0到($\sqrt{2}$)/2的数值
bias	一个常量	1

3. 使用Mahout构建模型

你可以利用*x*和*y*特征构建一个检测color字段的模型，使用如下命令即可：

```
$ bin/mahout trainlogistic --input donut.csv \
       --output ./model \
       --target color --categories 2 \
       --predictors x y  --types numeric \
       --features 20 --passes 100 --rate 50
...
color ~ -0.157*Intercept Term + -0.678*x + -0.416*y
      Intercept Term -0.15655
                   x -0.67841
                   y -0.41587
...
```

这条命令指定输入为资源中名为donut.csv的数据集，结果模型存放在文件./model中，目标变量在名为color的字段中且含有两个可能值。命令也指定了算法应该将变量x和y用作预测变量，二者都是数值型变量。余下的选项指定了学习算法的内部参数。

注意 你所看到的实际输出可能只有一位有效数字与此相同，因为该Logistic回归训练算法采用了某种随机化机制。

trainlogistic程序所有的命令行选项参见表13-7。

表13-7 trainlogistic程序的命令行选项

选　项	说　明
--quiet	产生较少的状态和进度输出
--input <file-or-resource>	使用指定的文件或资源作为输入
--output <file-for-model>	将模型存入指定的文件
--target <variable>	使用指定的变量作为目标
--categories <n>	指定目标变量的类别个数
--predictors <v1> ... <vn>	指定预测变量的名称
--types <t1> ... <tm>	给出了预测变量的类型列表。各个类型必须是数值、单词或文本。类型可以缩写为它们的第一个字母。如果给出的类型太少，就重复使用最后一个。用单词表示类别变量
--passes	指定训练过程中对入数据的复核次数。小规模的输入文件可能需要检验几十遍，很大的输入文件则可能不需要完全检查
--lambda	控制算法在最终模型中对变量的抑制程度。值为0表示不作为。典型的值在0.000 01或更小的数量级上
--rate	设定初始学习率。如果你有大量数据或设定了很高的复核次数，可以把它设大一点，因为它会随着数据核查的过程逐渐衰减
--noBias	消除模型中的截距项（一个内建的常数预测变量）。有时这会产生不错的效果，但通常并非如此，因为SGD学习算法一般能够在必要时消除截距项
--features	设定用于构建模型的内部特征向量大小。在这里较大的值会比较合适，尤其是处理文本型输入数据时

13.5.4 改进填充颜色分类器的性能

现在你已经训练好了第一个分类模型，下面来确定它在估计填充颜色的任务中性能如何。这个评估过程不仅仅是为项目评分，更为评估和改善分类器以达到最佳性能提供了一条途径。

1. 模型评估

注意，这个问题中填充点是完全被未填充点包围的，这意味着不可能用一个简单模型对点进行准确分类，比如SGD算法使用x和y坐标生成的模型。实际上，这个模型底层的线性方程只能产生一个负值，在Logistic回归的上下文中，保证不会产生大于0.5的分数 。我们将会看到，这第一个模型不是很有效。

模型训练好之后，你可以再次在训练数据上运行模型，以评估其表现（尽管我们知道它的表现不会太好）。

```
$  bin/mahout runlogistic --input donut.csv --model ./model \
        --auc --confusion
AUC = 0.57
confusion: [[27.0, 13.0], [0.0, 0.0]]
...
```

这里的输出包括两个我们特别感兴趣的值。首先是ACU值（Area Under the Curve，即曲线以下的面积，广泛用于评估模型质量），在这里为0.57。ACU的范围可以是从0（对应于一个完全错误的模型）到0.5（对应于一个随机猜测的模型），再到1.0（对应一个完全正确的模型）。此处，0.57这个值意味着我们的模型比随机猜测强不了多少。

要理解为什么这个模型的表现如此糟糕，你可以从混淆矩阵（confusion matrix）中找到线索。混淆矩阵是一个表，它比较实际结果与期望结果。所有得分低于默认阈值0.5的样本，都被划到未填充类别中。这使得分类器在2/3的情况下是正确的（40次里有27次正确），但仅在未填充数据上正确。虽然模型大部分时候都能得到正确答案，但这就好像一个停止的钟一天总能有两次指向正确的时间一样 。

注意　第15章会对混淆矩阵及其他度量工具做详细介绍。

runlogistic命令接受的选项参见表13-8。

表13-8　runlogistic程序的命令行选项

选　项	说　明
--quiet	产生较少的状态和进度输出
--auc	读入数据后打印模型在输入数据上的AUC分值
--scores	打印每个输入样本的目标变量值和分数
--confusion	打印某个阈值的混淆矩阵（参见--threshold）
--input <input>	使用指定的文件或资源作为输入
--model <model>	从指定文件中读入模型

2. 构建一个更有趣的模型

如果使用额外的变量进行训练，你会得到更有趣的结果。例如，下面的命令允许在建立模型时使用x和y，外加a、b和c来训练模型：

```
$ bin/mahout trainlogistic --input donut.csv --output model \
        --target color --categories 2 \
        --predictors x y a b c  --types numeric \
        --features 20 --passes 100 --rate 50
...
color ~ 7.07*Intercept Term + 0.58*x + 2.32*y + 0.58*a + -1.37*b + -25.06*c
      Intercept Term 7.06759
                   a 0.58123
                   b -1.36893
                   c -25.05945
                   x 0.58123
                   y 2.31879
...
```

注意，这个模型给了c变量很大的权重，而且截距项也较大。如果你忽略其他变量（虽然这么做不是非常合适，但截距项和c在这里占有绝对的主导权），模型线性部分的输出会在c=0时取到最大值11.5，一旦c>0.3时它就会迅速降为负数 。根据我们对这个问题的了解，这在几何上具有显著的意义。仅仅基于这一考量，就意味着这个模型与前面那个仅仅基于x和y、像停止的钟一样的模型不一样了。

Logistic 回归

Logistic 回归是一种分类模型，其中的预测变量的线性组合被传递给一个软限制函数，将输出限制在从 0 到 1 的范围内。Logistic 回归与其他一些模型密切相关，如感知机（软限制被替换为一个硬性限制）、神经网络（使用多层次的线性组合和软限制）和朴素贝叶斯算法（在独立假设的前提下，根据特征频率严格限制线性权重）。Logistic 回归不能分离所有可能的类别，但处理维度很高的问题时，你可以将一些预测变量进行组合引入新的变量，这都不是什么大问题。Logistic 回归在数学上的简洁性使算法的学习快速有效。

3. 再次测试

在训练数据上运行这个改进后的模型，你将得到一个与前面模型大不一样的结果：

```
$  bin/mahout runlogistic --input donut.csv --model model \
        --auc --confusion
AUC = 1.00
confusion: [[27.0, 0.0], [0.0, 13.0]]
entropy: [[-0.1, -1.5], [-4.0, -0.2]]
```

现在AUC值已经非常完美，达到了1，而且你可以从混淆矩阵中看到，模型已经能够正确分类所有的训练样本。

4. 用新数据进行测试

你可以在资源中的附加数据集donut-test.csv上测试同样的模型。因为这部分数据并未用于训练模型，这可能给我们带来一些惊喜。

```
$  bin/mahout runlogistic --input donut-test.csv --model model \
        --auc --confusion
AUC = 0.97
confusion: [[24.0, 2.0], [3.0, 11.0]]
entropy: [[-0.2, -2.8], [-4.1, -0.1]]
```


在这个留存数据集上，你可以看到AUC值降为0.97，但仍然很不错。混淆矩阵显示40个新样本中有5个分类错误，模型把2个未填充点标记为填充（误报），3个填充点标记为未填充（漏报）。显然，使用额外的变量，特别是c，大大改善了模型，但这里仍然存在一些问题。图13-10展示了所发生的事情。

注意图13-10中接近菱形底部的大实心圆。这个圆很接近训练集中的未填充颜色的样本，但没有与训练集中的任何填充颜色的样本相邻。因为学习算法不知道正确区域的形状如此奇怪，模型几乎不可能（在这个样本上）得到正确答案。

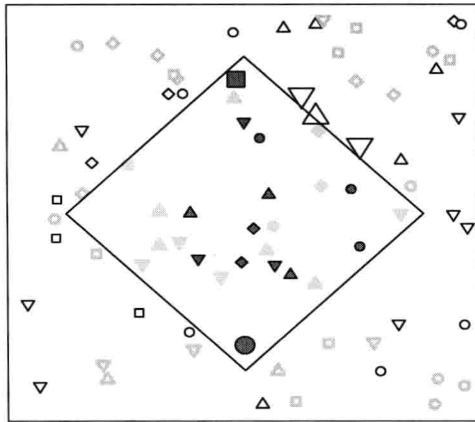


图13-10　在新数据集上试用分类器。原始数据显示为浅色阴影，新数据显示为填充或未填充。放大的符号为模型分类错误的点。菱形轮廓显示了我们实际用于生成甜面圈数据集的区域

5. 尝试其他模型

你可以选择其他的变量集合，来做一些额外的试验。例如，下面的代码将x、y、a和b包含到预测变量集中，但排除了c：

```
$ bin/mahout trainlogistic --input donut.csv --output model \
           --target color --categories 2 \
           --predictors x y a b  --types n --features 20 \
           --passes 100 --rate 50
...
color ~ 4.634*Intercept Term + -2.834*x + 5.558*y + -2.834*a + -6.000*b
      Intercept Term 4.63396
                   a -2.83439
                   b -5.99971
                   x -2.83439
                   y 5.55773
  ...
$  bin/mahout runlogistic --input donut-test.csv --model model \
             --auc --confusion
AUC = 0.91
confusion: [[27.0, 13.0], [0.0, 0.0]]
   entropy: [[-0.3, -0.4], [-1.5, -0.7]]
$
```

尽管这次模型并没有用上最合适的变量，但在留存数据集上的性能只是轻微下降。这表明，c所携带的信息也存在于x、y、a和b中。

你也可以在学习模型的参数上多做一些试验。这些实验是第15章中讨论的评估过程的开端。

13.6 小结

通过开发上一节中的示例程序，你已经用上了本章中关于分类前两个阶段的知识：训练模型，以及评估模型并调优性能。如果这是一个真正的系统，你的模型现在应该可以进入第三个阶段进行分类了：将其部署到真实的生产环境中。现在，你应该已经熟悉了分类的基本术语，而且对分类的概念以及工作方式有了比较扎实的理解。

记住，构建成功的分类器有一点很重要：用简单、具体的词项仔细地陈述问题，从而能够在一个预定义的类别（即目标变量）表中返回答案。注意，输入数据中并非所有特征都同等有效；你必须慎重选择用作预测变量的特征，而且需要尝试一些组合以便确认哪些有助于提高分类器的性能。记住，实践很重要。

掌握了这些基本思想，回过头来想想品酒的例子。对于基于机器的分类器，其目标应该是帮助你按时回家吃晚饭，而不是对生活中那些美好事物做出微妙的审美判断。

继续学习后面的第14章～第17章，你会发现Mahout为海量数据分类系统提供了强有力的工具，特别是当数据集规模超过100万样本时。第14章重点讨论如何从输入中提取特征以供学习算法使用，这是后面评估、调优训练好的分类器以在生产环境中部署分类器的第一步。

第 14 章

训练分类器

本章内容

- ❑ 提取文本中的特征
- ❑ 转换特征为Mahout所用
- ❑ 训练两个Mahout分类器
- ❑ 选择Mahout学习算法

本章探讨分类的第一阶段：模型训练。开发分类器是个动态过程，要求你创造性地思考出描述数据特征的最佳方式，并考虑在训练模型所选用的学习算法中如何使用这些数据特征。某些数据很容易就可以为分类所用，而有些则会给分类工作带来很大挑战，让你同时感受到沮丧、有趣和物有所值。

在本章中，你将学会挑选并有效地提取各种特征以构建Mahout分类器。特征提取所涉及的工作比第13章介绍的简化步骤多得多。我们将详细探讨特征提取，包括如何对原始数据进行预处理，将其变成可分类数据，以及如何将可分类数据变成适用于Mahout分类算法的向量。我们将以一个计算营销问题为例，演示如何从数据库中提取训练数据。

一旦理解如何为分类准备数据之后，我们将在14.4节给出一个示例，该示例利用Mahout中的随机梯度下降（SGD）算法在一个标准数据集20 Newsgroups上构建分类器。

在14.5节，我们将介绍Mahout分类中的各种学习算法的特点，这有助于了解如何根据具体项目的特点选择最合适的算法。在设计和训练分类器时，提取特征和选择算法这两项工作密切相关。为了培养对选择中不同做法的直观认识，我们给出第二个逐步介绍的例子，该例子使用另一种学习算法——朴素贝叶斯算法——对相同的数据进行处理。

下面我们先解释一下如何处理训练样本中的数据。

14.1 提取特征以构建分类器

把数据变成分类器可用的形式是一个复杂的过程，通常也很耗时。我们在这一节会大致介绍一下其中所涉及的工作。第13章中的图13-2展示了如何训练和使用分类模型，而图14-1是图13-2的一个简化视图。这里只需要一步，就可以从训练样本到达分类模型的训练算法。当然，现实情况会更加复杂。图14-1中还展示了第13章没提到的重要细节。

在图13-2中，训练数据到训练算法只用了一步。实际上，原始数据必须经过搜集和预处理，然后才能变成可分类的训练数据。图14-1中的训练样本是可分类数据，我们会在本章讨论从原始数据到可分类数据的处理过程，并在第15章和第16章给出进一步的细节。

原始数据经过预处理变为可分类形式之后，我们需要进行几步操作来选择预测变量和目标变量，并将它们编码为向量，即Mahout分类器所要求的输入形式。回想下第13章的内容，特征中可以用作预测变量的值有4种：

- 连续型；
- 类别型；
- 单词型；
- 文本型。

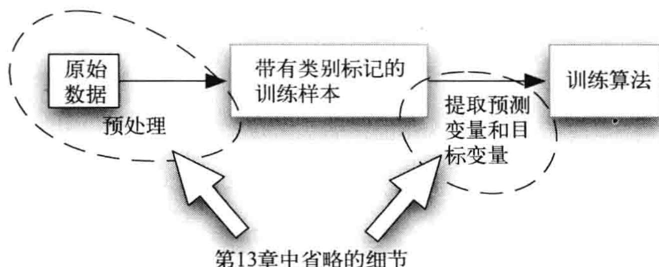


图14-1 图13-2的扩展部分。原始数据必须经过这些处理，然后才可以交付给训练算法

总体来说，这里对预测变量的描述是正确的，但它忽略了一个重要事实，即无论是在训练算法可以读取的内存还是文件格式中，将这些值交付给训练分类器的任何算法时，都必须以数字向量的形式表示。

与图14-1中经过简化的序列相比，图14-2中多了很多细节。为了将原始数据变成训练算法所要求的输入向量，要对其进行一系列的变换。这些变换可以分为两个阶段：预处理，产生用作训练样本的可分类数据；将可分类数据转换成向量。

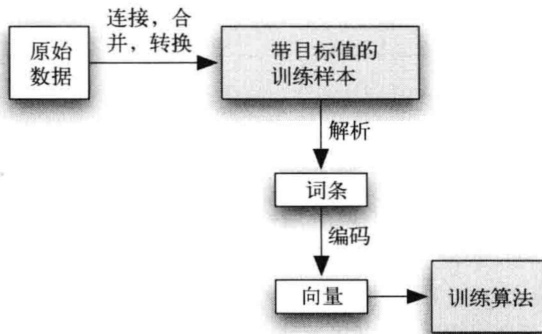


图14-2 向量是分类算法要求的输入格式。为了将数据编码为向量，搜索原始数据时必须要以可分类的单条记录形式来表示数据，以为解析和向量化过程做好准备

14

如图14-2所示，为训练算法准备数据主要包括两步。

(1) **原始数据的预处理**　经过重新组织，原始数据变成带有相同字段的记录。为了能够用于分类，这些字段可以是4种类型：连续型、类别型、单词型和文本型。

(2) **将数据转换为向量**　用自定义代码或者Lucene分析器、Mahout向量编码器之类的工具对可分类数据进行解析和向量化。有些Mahout分类器自己也包含向量化代码。

上述的第二步又可以分为两个阶段——词条化和向量化。对于连续变量而言，解析工作可能微不足道，但对于其他类型的变量，比如类别型、单词型或文本型变量，可能要涉及向量化操作。

本章的后续章节会详细讨论这两步处理。

14.2　原始数据的预处理

在特征提取的第一阶段，我们要重新认识一下数据，找出可以用作预测变量的特征。首先，根据分类目标选择目标变量，然后挑选或剔除特征，以得到值得一试的组合。这一步没有既定法则，全凭经验进行猜测，学习本章示例之后，你应该能慢慢积累出自己的经验。

本节简要概述了数据的预处理过程，包括搜集数据或重新将数据组织成单条记录，并从原始数据中提炼出第二层含义（比如将邮政编码转换为三数字编码或用生日来确定年龄）。在本章的示例当中，预处理并不是主要部分，这是因为这里用的数据集基本上已经做好数据提取了。而在第16章和第17章中的示例中，预处理扮演着更重要的角色。

14.2.1　原始数据的转换

在找出要尝试的特征之后，必须先把它们转换成可分类的形式。这个转换涉及将数据重新安排单一位置上并将其转换成合适的具有一致性的形式。

注意　可分类数据由具有相同字段的记录组成，字段的数据类型为下面4种之一：连续型、类别型、单词型或文本型。每条记录都包含一个训练样本的完全非规范化的描述。

乍一看，这一步好像不需要做就已经完成。如果数据看起来像单词，那特征肯定就是单词型，对不对？如果数据看起来像数字，那特征肯定是连续型，对吧？但我们在第13章已经讲过，第一印象可能会误导你。比如邮政编码，乍看像数字，但实际上是一个类别，是一个预先定义的类别的标签。包含单词的东西可能是单词型，或者最好看成类别型或文本型。用户ID或产品ID看起来可能像数值、类别或单词型数据，但更常见的做法为了支持它们所指向的用户或产品的特性对它们进行非规范化（denormalization）处理 。

下面的示例来自计算营销，我们可以将其作为为分类准备原始数据的练习。

14.2.2 一个计算营销的例子

假设你要构建一个分类模型，以确定用户是否会购买你向他们提供的某种产品。这算不上推荐系统，因为它是根据用户和产品的特性进行分类，而不是根据相似用户的集体行为（collective behavior）进行分类。

这个例子中的数据库中有几个数据库表，如图14-3所示。这些表是高度简化的零售系统的经典数据表。这里有用来表示用户和产品的表，有一个表记录展示或提供产品给用户的时间，还有一个表记录展示产品给用户导致的购买行为。这些数据目前还不能作为分类器的训练或测试数据，因为它们分散在几个表中。

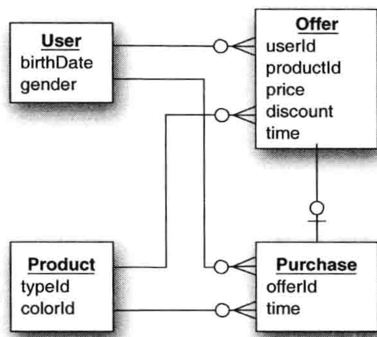


图14-3 包含不同数据类型的产品销售示例中的表结构。因为任何表中都没有分类器的训练样本记录，所以原始数据的这种组织形式不能直接用作训练数据

图14-3中展示的营销项目有很多种数据类型。用户有人口统计数据，比如生日和性别；产品有型号和颜色。向用户展示的产品数据记录在offer表中，跟offer相关联的产品购买记录放在purchase表中。

图14-4中是将这些数据变成分类器训练数据的一种可能方式。对于offer表中的每条记录，应该都有一条记录与之对应，但要注意是如何用产品和用户的ID来联结product和user表的。在这个过程中，用户的生日表示为年龄。我们用了一个外联结来推导产品展示到产品购买之间的时间延迟，并用一个标志来表明是否有购买行为发生。

为了将表中数据表示为图14-4中所示的可用形式，我们需要把它们集中到一起重新组织。为完成这个任务，可以像下面这样使用SQL查询：

```
select
  now()-birthDate as age, gender,
  typeId, colorId, price, discount, offerTime,
  ifnull(purchase.time, 0, purchase.time - offer.time) as purchaseDelay,
  ifnull(purchase.time, 0, 1) as purchased
from
  offer
  join user using (userId)
  join product using (productId)
  left outer join purchase using (offerId);
```

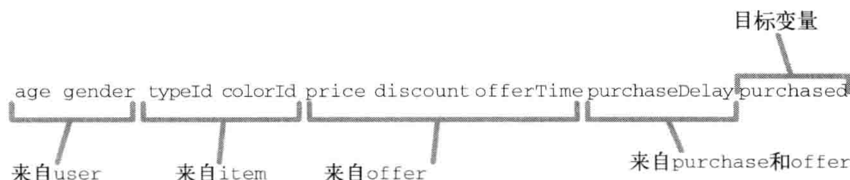


图14-4　为获得可分类数据，构建训练样本的数据来自不同的数据源，通过非规范化形成单条记录，来描述实际发生的情况。这里还对一些变量做了转换，比如用年龄表示生日，用延迟表示购买时间

该查询对存储在不同数据表中的数据进行了非规范化处理，将所有必要数据整合到一起形成记录。在这个例子中，offer表是主表，并且userId、productId以及offerId上的外键本质使得上述查询对offer表中的每条记录都恰好产生一条对应记录。需要注意的是跟purchase表的联结是外联结，这样就可以允许包含purchase.time的ifnull表达式在没有任何购买行为时产生0值。

注意　有时年龄更适合分类，而有时生日却更适合。比如说，在汽车事故的保险数据中，用年龄做变量会更好，因为与客户所属的时代相比，汽车事故跟客户年龄段的关系更大。而在购买音乐的数据中，生日可能更有意义，因为人们通常会保持自己早期对音乐的偏好，其对音乐的品味通常能折射出他们的时代特征。

上述查询语句产生的记录是可分类数据，可以用于训练算法的解析和向量化处理过程。向量化处理可以由你编码完成，或者把这些数据变成Mahout分类器可以接受的格式，因为这些分类器通常有自己的解析和向量化代码，可以帮你完成数据的解析和向量化处理。后续的内容和示例将重点讨论如何对解析后的可分类数据做向量化处理。

14.3　将可分类数据转换为向量

在Mahout中，Vector（向量）是一种保存浮点数字的数据类型，并且这些浮点值用整型做索引。本节会告诉你如何将数据编码为Vector，解释什么是特征散列（feature hashing），并演示Mahout API如何进行特征散列。我们还会看一下如何对不同类型的变量值进行编码。

前面有关聚类的各章中已经介绍过了Vector。很多种分类器，特别是Mahout中用的分类器，基本上都是以线性代数为基础，因此要求训练数据以Vector形式输入。

14.3.1　用向量表示数据

怎么用Vector表示可分类数据呢？表14-1中总结了几种办法。

表14-1 将可分类数据编码为向量的办法

方　法	优　点	代　价	用　途
每个单词、类别或连续值用一个Vector分量	没有冲突，易于实现逆处理	需要扫描两次[一次设置分量，一次设置值]，并且向量的长度可能不同	将Lucene索引导出为Vector用于聚类
将Vector隐式表示为词袋（bags of words）	扫描一次，没冲突	难以使用线性代数基元，难以表示连续值，并且必须将数据格式化为特殊的非向量形式	用在朴素贝叶斯中
用特征散列	扫描一次，向量大小提前固定，并适用于线性代数基元	有特征冲突，结果模型解释起来可能需要点技巧	在 OnlineLogisticRegression 和其他SGD学习算法中

Mahout中不同分类器使用了表14-1中的各种办法。我们来看一下如何将单词型、文本型和类别型值编码为向量。

1. 每个词一个分量

将可分类数据编码为Vector的一种办法是遍历两次训练数据：一次确定必需的Vector大小，并构建一个词典记录每个特征放在Vector中的什么位置；一次转换数据。这种方式可以用简单的表示来编码训练和测试样本：每个连续值、每个类别型、单词型和文本型数据中的独立单词或类别在向量表示中都会被分配唯一的位置。

这种方式明显的缺点在于要扫描两次训练数据，所以可能导致分类器的训练计算成本加倍，对于特别大的数据集来说这真是个问题。

Mahout中大多数聚类算法用的都是这种两次扫描的办法。

2. 将向量作为词袋

另外一种办法是包含特征名称，或是名称加上类别型、单词型或文本型变量值，而不是Vector对象。这一方法主要是用在朴素贝叶斯和补充朴素贝叶斯等Mahout分类器中。

这种办法的优势是可以不用词典，但这也意味着很难利用Mahout的线性代数功能，这些功能要求涉及的Vector向量长度已知并具有一致性。

3. 特征散列

基于SGD的分类器无需预先确定向量的大小，只要简单挑一个合理的大小，并把训练数据预置进那个大小的向量中。这种办法称为特征散列。预置时，我们通过连续变量变量名的散列，或者类别型、文本型或单词型数据的变量名和类别名或单词本身的散列，来选择一个或多个位置。

这种采用特征散列的办法有明显优势，需要的内存更少，也可以少扫描一次训练数据，但对向量进行逆向工程来确定映射到向量位置的原始特征也更加困难。这是因为多个特征可能会添加散列到同一个位置。当向量较大，或者每个特征对应多个位置时，这对于精确性不是什么问题，但可能会为理解分类器造成困难。

14.3.2　用Mahout API做特征散列

我们在这一节中看一下如何用Mahout API做特征散列。我们详细介绍如何对连续型、类别型、单词型和文本型特征编码。我们还解释特征冲突的概念以及它们对分类器的影响。

1. 对连续型特征编码

连续值是最容易编码的数据。连续变量的值可以直接加到为存储它们而分配的一个或多个位置上。这些位置是由特征的名称确定的。

Mahout对连续值的特征散列编码是通过`ContinuousValueEncoder`完成的。默认情况下，`ContinuousValueEncoder`只会更新向量中的一个位置，但你可以用`setProbes()`方法指定更新的位置数目。要编码变量的值，需要一个`Vector`，此时可以通过`new RandomAccessSparseVector(lmp.getNumFeatures())`建立一个新向量。然后，我们用`encoder.addToVector(value, vector)`对值编码。注意，Mahout假定被编码的值是字符串。如果被编码的值是`double`，一个`null`就会被传给`String`值，并且该值会被看成权重，即`encoder.addToVector(null, value, vector)`。

2. 对类别型和单词型特征编码

对具有*n*种不同值的类别特征编码，我们可以在*n*个向量位置中保存一个值`1`，标明变量取哪个值。为了降低冗余性，也可能会用*n*–1个位置，然后编码第*n*个类别的`1`值根本就不存（全是`0`）。

特征散列编码与此类似，但对特征名和值进行散列后才得到位置然后分散到整个特征向量中。此外，每个类别都可以与几个位置关联。特征散列的另外一个好处就是可以处理未知的和无限的单词型变量。

要用特征散列法对类别型或单词型变量编码，需要创建一个`WordValueEncoder`。这个编码器的用法和`ContinuousValueEncoder`是一样的，不过它默认探测数是2，而且被更新的位置是根据变量值及名称而相应变化的。有两种编码器：`AdaptiveWordValueEncoder`在运行过程中构建词典，以估计单词的出现频率，所以能给罕见词赋予较大的权重；`StaticWordValueEncoder`用已经建好的词典，如果没有词典，就给所有单词赋予相同的权重。

好的单词权重会对某些学习算法有极大的帮助，但也有一些，比如后面讲到的`OnlineLogisticRegression`类，所有单词都用相同的权重也没多大影响。

用相同的权重给单词编码类似于给连续值编码。下面是对连续变量编码的示例代码：

```
FeatureVectorEncoder encoder =
    new StaticWordValueEncoder("variable-name");

for (DataRecord ex: trainingData) {
  Vector v = new RandomAccessSparseVector(10000);
  String word = ex.get("variable-name");
  encoder.addToVector(word, v);
  // 使用向量
}
```

这段代码中构造了一个`StaticWordValueEncoder`，并将用来确定随机数生成器种子的名

称传给它。在内部，这个变量的名称和要编码的单词被组合起来，来得到编码操作要修改的位置，实现对这些值的编码。

向量的大小要通过实现来确定。比较大的向量会消耗更多的内存，并且训练过程也会变慢。而比较小的向量最终会导致太多的特征冲突，以致于学习算法无法弥补。

3. 对文本型特征编码

对文本型特征进行编码和对单词型特征进行编码类似，只不过文本中的单词很多。实际上，这里的例子只是简单地把文本中单词的向量表示叠加起来作为文本的向量。这种方法非常合适，但更细致的做法是先对单词进行计数，然后产生每个单词向量的加权总和，这样得到的结果更好。我们会在本章后续内容中讲解第二种做法。

提示　文本型数据值是单词的有序序列，但通常没必要把单词在文本中的顺序搞得太精确。只要把单词在文本中出现的次数搞清楚就够了，不用考虑顺序。话虽这么说，但通常文本的最佳向量并不是以单词出现次数为权重的单词向量总和，而是以单词出现次数的某种函数为权重。最常用的权重函数是平方根、对数，或不管单词出现的频率，只要出现就赋予相同的权重。代码清单14-1中的例子出于对简单性的考虑用了线性权重，但对于大多数应用程序而言，以数值的对数作为权重通常是更好的起点。

代码清单14-1展示了如何对文本中的所有单词进行编码，然后产生每个单词编码的线性权重之和，从而将文本编码为向量。这是用`StaticWordValueEncoder`实现的，并且还要有办法将文本分解或分析成单词。Mahout提供了编码器，Lucene提供了分析器。

代码清单14-1　文本的词条化和向量化

```
FeatureVectorEncoder encoder = new StaticWordValueEncoder("text");
Analyzer analyzer =
    new StandardAnalyzer(Version.LUCENE_31);          <—— 将文本分割为单词

StringReader in = new StringReader("text to magically vectorize");
TokenStream ts = analyzer.tokenStream("body", in);
TermAttribute termAtt = ts.addAttribute(TermAttribute.class);

Vector v1 = new RandomAccessSparseVector(100);       <—— 编码进大小为100的向量
while (ts.incrementToken()) {
  char[] termBuffer = termAtt.termBuffer();
  int termLen = termAtt.termLength();

  String w = new String(termBuffer, 0, termLen);
  encoder.addToVector(w, 1, v1);                     <—— 将单词w加到向量v中
}
System.out.printf("%s\n", new SequentialAccessSparseVector(v1));
```

这段代码会产生向量的可输出形式：

```
{8:1.0,21:1.0,67:1.0,77:1.0,87:1.0,88:1.0}
```

这个输出的图形化形式如图14-5所示。

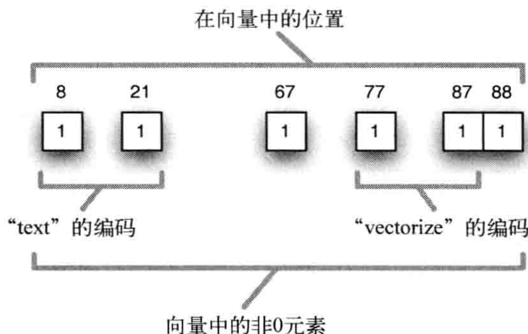


图14-5 对“text to magically vectorize”编码的向量。这个向量中有6个非0值，向量大小为100。等于0的值没有保存，因此也没像Lucene标准分析器产生的结果那样显示出来

如图14-5所示，位置8和21是单词text的编码，77和88是vectorize的编码，67和88是magically的编码。单词to被Lucene标准分析器去掉了。这一过程的最终结果是个稀疏向量，其中的0值根本不会保存。

4. 特征冲突

数据的散列特征表示可能会把不同的变量或单词存在相同的位置，从而引发冲突。比如说，如果你在前面的例子中用了一个长度为20，而不是100的向量，那对“text to magically encode”编码的结果值可能是这样的：

```
{1:1.0,7:2.0,8:2.0,17:1.0}
```

在这个向量中，位置7和8的值不再是1，变成了2，像图14-5中显示的所有非0单元一样。这个向量如图14-6所示。

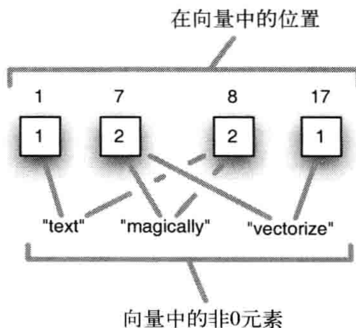


图14-6 当向量的长度不再是100，变成20时，冲突出现了。单词text和magically在位置8上冲突了，magically和vectorize在位置7上冲突了。这些冲突使结果向量更难解释，但通常它们不会影响分类的准确性

处理特征冲突以避免对性能造成影响非常简单。在图14-6显示的向量中，单词text被编码在位置1和8，magically在位置7和8，vectorize在位置7和17。也就是说，text和magically共享了位置8，

vectorize和magically共享位置7。所以位置7和8的值是2，而其他非0值都是1。因为每个单词都被赋予了两个位置，所以学习算法能通过学习弥补这些冲突。

在这个例子中，学习算法可以弥补这些冲突，因为从单词到向量的转换是可逆的。通常不太可能每个单词都有单一位置，因为向量的维度几乎总是比文本中的词汇量小。尽管有多个位置，这种向量化技术通常都很管用，其原因跟布隆过滤器能用的原因一样。

两个单词之间在哪里出现冲突对于分类器来说有很大的差别，这两个单词不太可能在别的位置上也发生冲突，也不太可能和第三个单词发生冲突。因此，通常这种向量化方式很好用，并且如果向量长度选得足够大，不会有明显的准确率损失。找出确切的向量长度要用到第16章中的评估技术，通过评估可以进行实证性验证。

接下来，我们会用SGD Mahout分类算法在一些真实数据上尝试这些办法。（14.5节会讨论不同的Mahout分类算法，14.6节会用另一种Mahout算法对相同的数据集进行分类。）

14.4 用 SGD 对 20 Newsgroups 数据集进行分类

我们在这一节用SGD学习算法为20 Newsgroups数据集构建一个分类模型。在这个例子中，我们会对数据进行预览，并做初步的分析，分析哪些是最常见的文件头，通过解析和词条化处理将数据转换为向量，并为20 Newsgroups数据集项目编写训练代码。

注意 20 Newsgroups数据集是机器学习研究中常用的标准数据集。这些数据是20世纪90年代早期20 个Usenet新闻组上几个月消息的副本。

这个例子重点强调特征提取，并且集中关注特征提取的第二阶段，即向量化处理。之所以用20 Newsgroups数据集，这是因为对这些数据做特征提取的第一阶段，即将原始数据转换为可分类数据的预处理过程相对简单——创建该数据集的研究人员已经完成了大部分工作。

14.4.1 开始：数据集预览

准备数据集的第一步就是检查数据，并确定哪些特征可能有助于将样本分到选定目标变量的类别中。（在这里，目标变量就是20个新闻组中的每一个。）

首先，从http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz处下载20 Newsgroups数据集。根据日期，该版本数据集被划分成训练数据和测试数据，并保留了所有数据的文件头（header line）。在真实场景中，分类器一般都用来处理新数据，而非旧数据，按日期划分的好处就在于它更贴近真实场景。

如果你查看训练数据目录下的某个文件，比如20newsbydate-train/sci.crypt/15524，应该能看到下面这种内容：

```
From: rdippold@qualcomm.com (Ron "Asbestos" Dippold)
Subject: Re: text of White House announcement and Q&As
```

14

```
Originator: rdippold@qualcom.qualcomm.com
Nntp-Posting-Host: qualcom.qualcomm.com
Organization: Qualcomm, Inc., San Diego, CA
Lines: 12

ted@nmsu.edu (Ted Dunning) writes:
>nobody seems to have noticed that the clipper chip *must* have been
>under development for considerably longer than the 3 months that
>clinton has been president. this is not something that choosing
...
```

20 Newsgroups数据集是由消息组成的，每个文件一条。每个文件都从文件头开始，指明了消息是由谁发送的，有多长，用的什么软件，以及消息的主题是什么之类的信息。接着是一个空白行，然后是无格式文本的消息体。

这种数据的预测特征或者在文件头中，或者在消息体中。所以先检查所有文档中文件头的不同字段出现的次数。这可以帮我们确定哪些是最常用因此很可能影响许多文档的分类结果的字段。

因为这些文档格式简单，像下面这种bash脚本就能统计出各种文件头的出现次数：

```
#!/bin/bash
export LC_ALL='C'
for file in 20news-bydate-train/*/*
do
  sed -E -e '/^$/,$d' -e 's/:.*//' -e '/^[[:space:]]/d' $file
done | sort | uniq -c | sort -nr
```

这个脚本会扫描训练数据集中的所有文件，发现第一个空白行后就把后面的所有文本删除，剩余行中冒号后面的内容也会删掉。此外，文件头中以空格开头接续上一行内容的文本行也会删掉。然后，它对sed中剩下的输出进行排序、计数，然后再根据计数的结果按降序进行排序。

训练样本中文件头的计数汇总如表14-2所示。标记为“...”的那一行表明，我们跳过了中间行，直接展示列表最后那些稀奇古怪的例子。“动作”列表明根据文件头的潜在价值可能要执行哪些操作。可选的操作是在解析过程中抛弃没价值的文件头（丢弃），留下可能有用的（保留），或者尝试价值不太明确的（尝试？）。作用尚不明确的数据很值得一试，因为最终可能会证明它非常有价值。也许我们应该把它们标记为“一定要试一下！”。

表14-2　20 Newsgroups文章中最常见的文件头。不过，其中也有一些不太常见的

文件头	计数	备注	动作
标题	11 314	文本	保留
源自	11 314	消息的发送者	保留
行数	11 311	消息的行数	保留
组织	10 841	跟发送者相关	尝试?
分发	2533	潜在的目标泄漏	尝试?
Nntp发帖主机	2453	跟发送者相关	尝试?
NNTP发帖主机	2311	跟上面那个一样，只是变成了大写	尝试?

（续）

文件头	计数	备注	动作
回复	1720	可能是个线索，不太常见，可以试验一下	尝试?
关键词	926	内容描述	保留
文章-I.D.	673	太具体了	丢弃
X-Newsreader	588	发送者用的软件	丢弃
总结	391	跟“关键词”类似	保留
始发	291	跟“源自”类似，但不太常见	丢弃
在回复中	219	很可能只是个ID，很少见	丢弃
新闻软件	164	发送者用的软件	丢弃
过期时间	113	具体日期，很少见	丢弃
在回复中	101	很可能只是个ID，很少见	丢弃
致	80	潜在的目标泄漏，很少见	丢弃
X-Disclaimer	64	噪声，很少见	丢弃
免责声明	56	因为跟新闻组的偏好程度有关，可以作为潜在的线索，很少见	丢弃
…		…	
天气	1	奇怪的文件头	丢弃
Orginization	1	文件头中出现拼写错误很奇怪	丢弃
Oganization	1	好吧，可能不是	丢弃
Oanization	1		丢弃
Moon-Phase（月相）	1	也是，有个帖子中真有这个；开源就是这么精彩	!?!

很多文件头可能都没什么价值，也有很多出现的次数太少，所以基本没什么影响。表14-2把Subjects、From、Lines、keywords和Summary作为可能感兴趣的特征文件头。它们经常出现，而且看起来很可能跟文档的内容相关。其中比较奇怪的是Lines，实际上它是和内容相关的，因为有些新闻组可能习惯于发比较长的文档，而有些则一般发比较短的。

提示　数据的初步分析对于分类能否成功至关重要。有时候这种分析很有意思，会有彩蛋出现，比如表14-2中的Moon-Phase文件头。这些惊喜对于构建分类器可能也很重要，因为它们经常能揭示数据中的问题，或让你产生很关键的见解，从而简化分类问题。要尽早可视化，并且要经常可视化。

注意，20 Newsgroups示例中的数据是精心准备的，所以很容易用于测试分类器。因此，它有些理想化了。因为所有数据都在一起，并且所有明显的目标泄漏都已经被去掉了，所以你不需要对它们进行预处理，可以直接进行文本分析。

14.4.2　20 Newsgroups数据特征的解析和词条化

为学习算法准备数据的第二阶段是将可分类数据（四种可能类型）转换为向量。在20 Newsgroups数据集中，除了Lines，所有数据字段都是文本型或单词型，其格式看起来用标准的Lucene词条化工具就可以轻松完成词条化。Lines字段是数字，也能用Lucene词条化工具解析。看起来这个字段对于区分某些新闻组非常有价值，因为talk.politics.*小组每条消息的平均行数约为60，而misc.forsale中文档的平均行数只有25。

14.4.3　20 Newsgroups数据的训练代码


No. 13

现在可以建模型了。为了简单起见，我们在这个例子中要用前面对点进行分类时用过的OnlineLogisticRegression算法。但这一次是要从Java代码中运行分类器，而不是用命令行，所以能看到提取特征和向量化的过程。

Logistic回归分类器一览

Logistic回归分类器将输入值进行线性组合后用Logistic函数$1/(1+e^{-x})$将值压缩到(0, 1)区间之内。Logistic回归模型的输出一般都可以解读为概率估算值。此外，即便特征向量的维度很高，线性组合中所用的权重也可以通过增量方式高效计算。因此Logistic回归经常用于可扩展的串行学习中。Logistic回归接受的特征必须为数字形式，所以文本、单词和类别型变量值必须编码成向量格式。

1. 建立向量编码器

首先需要有对象把文本和行数转成向量值，代码如下所示：

```
Map<String, Set<Integer>> traceDictionary =
  new TreeMap<String, Set<Integer>>();
FeatureVectorEncoder encoder = new StaticWordValueEncoder("body");
encoder.setProbes(2);
encoder.setTraceDictionary(traceDictionary);
FeatureVectorEncoder bias = new ConstantValueEncoder("Intercept");
bias.setTraceDictionary(traceDictionary);
FeatureVectorEncoder lines = new ConstantValueEncoder("Lines");
lines.setTraceDictionary(traceDictionary);
Dictionary newsGroups = new Dictionary();
Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_31);
```

在这段代码中，3类数据分别由3种编码器处理。第一个编码器encoder用来对帖子标题和主体内容中的文本编码。第二个编码器bias提供了一个常数偏移量，模型可以用它于对每类的平均频率进行编码。第三个是lines，用来对消息的行数编码。

因为OnlineLogisticRegression类在训练过程中希望得到目标变量的整型ID，所以还需要用一部词典把目标变量（新闻组）转成整型值，该词典就是newsGroups对象。

2. 配置学习算法

可以像下面这样配置Logistic回归学习算法：

```
OnlineLogisticRegression learningAlgorithm =
    new OnlineLogisticRegression(
          20, FEATURES, new L1())
        .alpha(1).stepOffset(1000)
        .decayExponent(0.9)
        .lambda(3.0e-5)
        .learningRate(20);
```

学习算法构造函数接受的参数包括：指定的目标变量类别的个数、特征向量的大小及正则化项（regularizer）。此外，学习算法中还有一些配置方法。其中，`alpha`、`decayExponent`和`stepOffset`方法可以指定学习率及衰减率和衰减方式。`lambda`方法指定正则化的权重，`learningRate`方法指定初始学习率。

在生产模型中，学习算法需要运行成百上千次才能找到比较理想的取值。为了确定比较合理的取值，我们在这里做了几个快速试验。

3. 访问数据文件

接着需要得到所有训练数据文件的清单，代码如下所示：

```
List<File> files = new ArrayList<File>();
for (File newsgroup : base.listFiles()) {
  newsGroups.intern(newsgroup.getName());
  files.addAll(Arrays.asList(newsgroup.listFiles()));
}

Collections.shuffle(files);
System.out.printf("%d training files\n", files.size());
```

这段代码把所有新闻组的名字都放到词典中。像这样预定义的词典内容，可以确保词典中的条目是以稳定并且可识别的顺序存放的。这有助于在多次训练过程运行结果之间进行比较。

4. 数据词条化前的预备工作

这些文件中的大部分数据都是文本型，因此可以使用Lucene来做词条化处理。使用Lucene会比基于空格或标点符号的简单分割要好，这是因为Lucene的`StandardAnalyzer`类能够正确地处理一些特殊词条，如电子邮件地址。

你也需要多个变量来累积运行过程中的平均值，这些变量包括平均对数似然、正确率、文档行数和处理的文档数目等：

```
double averageLL = 0.0;
double averageCorrect = 0.0;
double averageLineCount = 0.0;
int k = 0;
double step = 0.0;
int[] bumps = new int[]{1, 2, 5};
double lineCount;
```

这些变量有助于度量学习算法的进度和性能。

5. 读取数据并进行词条化处理

经过前面那些准备工作，现在你已经可以处理数据了。因为在线Logistic回归算法学得很快，所以我们只需要扫描一遍数据。只扫描一次对进度评估也有好处，因为在将文档用作训练数据之前，可以用当前状态的分类器逐一对它们进行测试。

在真正的学习过程中，文档是以随机顺序处理的，所以来自不同新闻组的样本会混在一起。将训练数据以随机顺序交付给分类器，可以使OnlineLogisticRegression算法更快收敛到某个结果。具体做法如下所示：

代码清单14-2　解析数据

```
for (File file : files) {
  BufferedReader reader = new BufferedReader(new FileReader(file));
  String ng = file.getParentFile().getName();                        ❶ 识别新闻组
  int actual = newsGroups.intern(ng);
  Multiset<String> words = ConcurrentHashMultiset.create();

  String line = reader.readLine();
  while (line != null && line.length() > 0) {                        ❷ 检查行数文件头
    if (line.startsWith("Lines:")) {

      String count = Iterables.get(onColon.split(line), 1);
      try {
        lineCount = Integer.parseInt(count);
        averageLineCount += (lineCount - averageLineCount)
            / Math.min(k + 1, 1000);
      } catch (NumberFormatException e) {
        lineCount = averageLineCount;
      }
    }
    boolean countHeader = (
        line.startsWith("From:") || line.startsWith("Subject:")||
        line.startsWith("Keywords:")|| line.startsWith("Summary:"));
    do {
      StringReader in = new StringReader(line);
      if (countHeader) {                                             ❸ 计算文件头中单词的数量
        countWords(analyzer, words, in);
      }
      line = reader.readLine();
    } while (line.startsWith(" "));
  }                                                                  ❹ 计算内容主体中单词的数量
  countWords(analyzer, words, reader);
  reader.close();
}
```

用文件名判断每篇文档所属的新闻组❶。文件头中有一行给出了文档的行数，而行数可以编码为特征❷。然后，解析文件头❸和文档的主体❹，我们需要对找到的单词计数。这里可以用Google Guava类库中的multi-set来做计数工作。这些文档中至少有一篇的行数是假的，在这种情况下，为保险起见，将行数设置为总体的均值是一个比较合理的做法。

每篇文档都是以文件头开始的，文件头的每一行都是由文件头名、分号、内容组成，有些文件头可能不止一行，接续的内容换行后前面会有一个空格。行数以及选定文件头行的单词数要作为特征处理。在文件头处理完后，文档的主体要经过Lucene分析器（analyzer）的处理。

6. 数据的向量化

得到文档中的数据之后，你可以将所有特征收集到一个特征向量中，以供分类器的学习算法使用。下面是完成这一任务的代码：

```
Vector v = new RandomAccessSparseVector(FEATURES);
bias.addToVector(null, 1, v);
lines.addToVector(null, lineCount / 30, v);
logLines.addToVector(null, Math.log(lineCount + 1), v);
for (String word : words.elementSet()) {
  encoder.addToVector(word, Math.log(1 + words.count(word)), v);
}
```

首先，对bias（常量）进行编码，它的值始终是1。学习算法可以利用该特征作为阈值。如果没有这样一个bias，有些问题就没办法用Logistic回归解决。

行数同时编码为原始形式和对数形式，除以30可以把行长度放入和其他输入几乎一样的区间，以便加快学习进程。

文档主体的编码与此类似，但文档中每个单词的权重依据单词在文档中出现频率的对数来确定，而不是直接使用频率。这样做是因为单词在单篇文档中出现多次的频率，要比单词出现的预期整体频率要高。正是基于这种考虑，我们使用了频率的对数。

7. 评估当前进度

此时，你应该拥有了一个可以交付给学习算法的向量。直到现在，我们仍然可以认为文档中的数据没有完全被提炼出来。因此，可能你还可以用这些数据得到一些关于分类器准确度的反馈，甚至有望做出改善。这里有两个可以衡量准确性的指标：对数似然值和正确分类的平均比例。

对数似然值最大为0，在对20种候选答案进行随机猜测时，其结果应该接近–3。即便分类器给出错误答案，但只要正确答案的排名靠前，对数似然也会给一些分值。而即使分类器给出正确答案，但如果有一个错误答案的得分几乎和该正确答案一样高，对数似然就会减掉一些分值。对数似然值这种但求近似不求完全命中的特性，使得它特别适合作为性能测试的指标。但跟没有技术背景的同事解释对数似然也实在让人头疼，所以我们还是需要更简单的指标，即得出正确答案的平均比例。

我们可以用Welford算法来计算性能指标的均值，这种算法的好处在于总能得到当前均值的估值。我们这里用的是Welford算法的变体，因此只有在处理不足200个样本之前，才会用直接平均值。而在那之后用的是指数平均值，这样最终得到的进度性能指标就可以忽略掉分类器早期还没学会任何东西时给出的结果。

下面这段代码就是计算对数似然值和正确平均百分比的：

```
double mu = Math.min(k + 1, 200);
double ll = learningAlgorithm.logLikelihood(actual, v);
averageLL = averageLL + (ll - averageLL) / mu;

Vector p = new DenseVector(20);
learningAlgorithm.classifyFull(p, v);
int estimated = p.maxValueIndex();

int correct = (estimated == actual? 1 : 0);
averageCorrect = averageCorrect + (correct - averageCorrect) / mu;
```

在这段代码中，模型需要得出一些性能指标。它计算对数似然值的均值并放在变量

averageLL中。然后，它用比例最高的新闻组确定变量estimated，并与正确的值进行比较。然后，对比较结果取均值，得出正确结果的平均百分比，作为变量averageCorrect的值。

8. 用编码数据训练SGD模型

从当前训练样本中得到进度信息后，我们可以把它传给学习算法以更新模型。如果学习算法多次扫描数据，那么最好将测试样本用于进度监控，将训练样本只用于训练，而不是像这里一样将两种样本都用于训练和进度监控。

在示例代码中，我们施展一些“花式步法”，以逐步增长的时间间隔来提供进度反馈。这样在运行过程中就可以尽早提供反馈，又不至于因为程序运行时间太长让你淹没在数据洪流之中。下面的代码给出了这些逐步增长的步长的计算过程：

```
learningAlgorithm.train(actual, v);
k++;
int bump = bumps[(int) Math.floor(step) % bumps.length];
int scale = (int) Math.pow(10, Math.floor(step / bumps.length));
if (k % (bump * scale) == 0) {
  step += 0.25;
  System.out.printf("%10d %10.3f %10.3f %10.2f %s %s\n",
       k, ll, averageLL, averageCorrect * 100, ng,
       newsGroups.values().get(estimated));
}
learningAlgorithm.close();
```

样本数量每次达到bump * scale，都会新输出一行学习算法当前的状态。随着学习不断进行，状态报告的频率会逐步递减。也就是说，准确性的变化越快，状态的更新越频繁。

最后一句是告诉学习算法可以收工了。这对所有被延迟的学习生效，并将一切临时结构清除干净。

14.5 选择训练分类器的算法


No. 14

Mahout的主要优势在于它处理超大并一直增长的数据集时所体现出来的健壮性。Mahout中的所有算法都具备扩展能力，但它们在其他方面各有特色，能在不同的情景下发挥各自的特长。表14-3对Mahout内部用于分类的不同算法进行了比较。这张表及本节余下的内容，有助于你确定哪个Mahout算法最适合特定的分类问题。但你要记住，这里罗列出来的并不是Mahout的全部算法，因为总有新算法被不断开发出来。

表14-3 Mahout中用于分类的学习算法

数据集大小	Mahout算法	执行模型	特 性
小到中型（训练样本数在千万以内）	随机梯度下降（SGD）一族：OnlineLogisticRegression、CrossFoldLearner、AdaptiveLogisticRegression	串行，在线，增量式	使用全部类型的预测变量，在数据规模合适（上至几百万训练样本）的情况下十分适合、高效
	支持向量机（SVM）	串行	仍处于实验阶段，在数据规模合适的情况下十分适合、高效

（续）

数据集大小	Mahout算法	执行模型	特　性
中到大型（训练样本数在百万到上亿之间）	朴素贝叶斯	并行	特别偏爱文本型数据；需要中等到很大的训练开销；处理那些对于SGD或SVM来说过大的数据集实用有效
	补充朴素贝叶斯	并行	比朴素贝叶斯的训练成本高一些；处理对于SGD来说过大的数据集实用有效，但有和朴素贝叶斯类似的局限性
小到中型（训练样本数在千万以内）	随机森林	并行	使用全部类型的预测变量；训练开销高；尚未得到普及；成本高，但能实现复杂而有趣的分类，比其他技术更擅于处理数据中非线性和条件关系

这些算法的差异体现在训练的开销或成本，性能表现最好时对应的数据集规模，以及能够实现的分析的复杂度等方面。

14.5.1　非并行但仍很强大的算法：SGD和SVM

正如在图13-1和图13-7中看到的那样，即便算法不是并行的，其行为也会有很强的扩展能力。这一节概述两个串行执行的Mahout学习算法：随机梯度下降（SGD）和支持向量机（SVM）。

1. SGD算法

SGD算法应用广泛，是那种靠每个训练样本对模型进行微调，然后逐步接近该样本正确答案的学习算法。这一递增模式在多个训练样本上重复执行。我们可以借助一些特殊技巧来决定对模型的微调程度，使模型仅在一定数量的样本上训练之后能准确对新数据进行分类。尽管很难让SGD算法实现高效的并行化处理，但因为它们处理大多数应用时一般都很快，所以也没必要并行执行。

因为这些算法对每个训练样本执行相同的简单操作，所以它们所需的内存大小是恒定的，这一点很重要。出于这个原因，每个训练样本所需的工作量基本一致。这些特性使基于SGD的算法的性能是线性的，即处理两倍的数据仅需两倍的时间。

2. SVM算法

Mahout最近新加入了一个SVM算法的实验性串行实现。该实现包括用Java实现的LIBLINEAR类库，具备工业级强度但不可扩展，该类库之前只有C++版。该SVM实现仍然是个新东西，在部署前应该认真测试。

SVM算法的表现跟SGD很像，都是串行实现，但对于大量数据的训练速度可能比SGD还要慢一些。Mahout的SVM实现很可能分享了SGD的输入灵活性和线性扩展能力，所以对于中等数据规模的项目来说可能优于朴素贝叶斯。

14.5.2　朴素分类器的力量：朴素贝叶斯及补充朴素贝叶斯

如表14-3所示，Mahout中的朴素贝叶斯和补充朴素贝叶斯算法都是并行算法，在实际应用中，

比基于SGD的算法更适合处理大型数据集。因为它们可以同时在多台机器上高效工作，所以这些算法能处理非常大的数据集，而基于SGD的算法在这方面则略逊一筹。

然而，Mahout实现的朴素贝叶斯，仅限于基于单一文本型变量进行分类。对于很多问题来说，包括典型的大规模数据问题，这都不是问题。但如果需要连续变量，并且不能将其量化为单词型对象从而和其他文本数据一块处理，可能就没办法使用朴素贝叶斯一系的算法。

此外，如果数据中有不止一类的单词型或文本型变量，可能需要把这些变量拼接到一起，并以一种明确的方式添加前缀以消除歧义。这样做可能会损失重要的差异信息，因为所有单词和类别的统计数据都混到一起了。但大多数文本分类问题，应该都可以用朴素贝叶斯或补充朴素贝叶斯算法解决。

提示　如果你要处理上千万的训练样本，并且预测变量只有单个文本型值，那么朴素贝叶斯或补充朴素贝叶斯可能是最理想的算法。而对于其他类型的变量，或者训练数据没这么多的话，可以试试SGD。

如果Mahout朴素贝叶斯的限制条件与你的问题吻合，那它们就是首选的算法。它们擅长处理超过100 000训练样本的数据，并且在处理超过千万的训练样本时，很可能会优于串行执行的算法。

14.5.3　精密结构的力量：随机森林算法

Mahout中有Leo Breiman的随机森林算法的串行和并行两种实现。这一算法首先训练大量的简单分类器，然后通过投票机制得出最终的唯一结果。Mahout的并行实现是在模型中并行训练很多分类器。

上述并行方式具有不太寻常的扩展性质。因为每个小分类器都是在所有训练样本上针对部分特征训练，于是集群中每个节点对内存的要求大致与训练样本数目的平方根成正比。这一点就不像朴素贝叶斯那样好，后者对内存的需求跟看到的独立单词数目成正比，因此大约是跟训练样本数目的对数成正比。

但这种不太理想的扩展性质也会带来“回报”，在处理那些对Logistic回归、SVM或朴素贝叶斯来说比较困难的问题时，随机森林模型有它的独到之处。一般而言，这种问题都需要模型用变量的相互作用和离散化来处理连续变量的阈值效应。比较简单的模型经过足够的时间和变量变换工作之后，也能处理这些效应，但随机森林通常不需要做这些工作就能解决这些问题。

现在你已经了解Mahout为训练分类器所提供的学习算法了，下面该用实践来检验你所学的知识了。在14.4节，我们已经用SGD算法针对20 Newsgroups数据训练过一个分类器；下一节，我们要用另一种算法试一下，你也可以比较一下这两种方法的结果。

14.6 用朴素贝叶斯对 20 Newsgroups 数据分类

分类路径部分取决于所用的Mahout分类算法。正如前面一节提到的，算法是并行还是串行执行有很大差别。并行的SGD算法和串行的朴素贝叶斯算法有不同的输入路径，而这种差异又要求用不同的方式处理数据，特别是向量化。

本节会向你展示如何用朴素贝叶斯模型处理14.4节处理过的20 Newsgroups数据。对相同的数据用不同的算法，你就能看出串行方式和并行方式在构建分类模型上的差异。

我们首先要为训练算法准备数据，进行数据提取，然后你将了解如何训练模型。完成之后，你就可以开始评估最初的模型，并确定它是表现良好，还是需要进行调整。

14.6.1 开始：为朴素贝叶斯提取数据

我们先把数据转换成可分类形式，并转成朴素贝叶斯算法用的文件格式。所用的数据就是14.4节给SGD示例所用的20 Newsgroups数据集。

朴素贝叶斯分类器可以和跟SGD分类器一样以编程方式驱动，但它也有可以通过命令行调用的内置解析器。这个解析器可以接受SVMLight程序所用的数据文件格式的一种变体，其中的每条数据记录在文件中占一行，包括目标变量的值，后跟用空格分隔的特征，出现特征名表示该特征的值为1，没有则表明该特征的值为0。要用命令行版本的朴素贝叶斯分类器，你必须将20 Newsgroups数据集中的数据转换成这种格式。

在20 Newsgroups数据集中，每个新闻组（newsgroup）的数据占一个目录，目录中的每个文件都是一个文档。我们需要扫描所有的目录，并把文件都转成单行文本，以目录名开头，跟着是文档中出现的所有单词。Mahout中的`prepare20newsgroups`程序完成的就是这个功能。要转换训练和测试数据，请用下面的命令：

```
$ bin/mahout prepare20newsgroups -p 20news-bydate-train/ \
       -o 20news-train/ \
       -a org.apache.lucene.analysis.standard.StandardAnalyzer \
       -c UTF-8

no HADOOP_CONF_DIR or HADOOP_HOME set, running locally
INFO: Program took 3713 ms

$ bin/mahout prepare20newsgroups -p 20news-bydate-test \
       -o 20news-test \
       -a org.apache.lucene.analysis.standard.StandardAnalyzer \
       -c UTF-8
no HADOOP_CONF_DIR or HADOOP_HOME set, running locally
INFO: Program took 2436 ms
```

命令中的选项`-p`指定存放训练或测试数据的目录名，选项`-o`指定输出目录，选项`-a`指定文本解析器，选项`-c`指定将输入字节转成文本所用的字符编码类型。

这个命令的运行结果应该是个叫做20news-train的目录，每个新闻组一个文件。在像20news-train/misc.forsale.txt这样的数据文件中，你会见到如图14-7所示的结果。

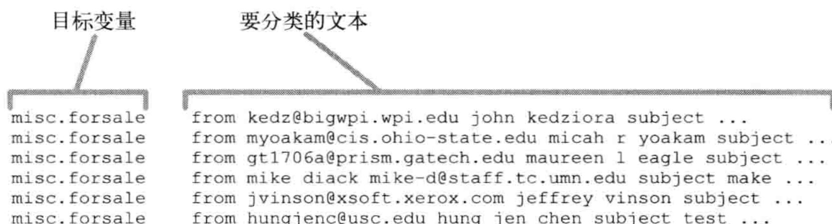


图14-7　将训练数据转换为朴素贝叶斯程序偏爱的格式。注意，这里显示的内容已经缩短了很多

14.6.2　训练朴素贝叶斯分类器

至此为止，我们一直在处理20 Newsgroups数据，提取特征为朴素贝叶斯算法准备恰当的输入数据。将训练和测试数据转换成正确的格式后，可以训练分类模型了。

试一下下面的命令，产生一个使用朴素贝叶斯算法的训练模型：

```
bin/mahout trainclassifier -i 20news-train \
            -o 20news-model \
            -type cbayes \
            -ng 1 \
            -source hdfs
...
INFO: Program took 250104 ms
```

运行结果是存在20news-model目录中的模型，这个目录是由选项-o指定的。选项-ng表明考虑独立的单词而不是单词的短序列。模型中有几个文件，分别包含模型的一部分。这些都是二进制文件，想直接检查不太容易，但你能借助testclassifier程序用它们对测试数据进行分类。

你现在已经利用20 Newsgroups数据和朴素贝叶斯算法构建一个模型，并进行了初步的训练。这个模型好用吗？你可以评估模型的性能，以回答这个问题。

14.6.3　测试朴素贝叶斯模型

现在是新训练模型的评估时间。我们会介绍评估流程，并对新训练的模型进行一些初步的测试。在第15章，我们会深入探讨这一主题。

要在测试数据上运行朴素贝叶斯模型，可以使用下面的命令：

```
bin/mahout testclassifier -d 20news-test \
            -m 20news-model \
            -type cbayes \
            -ng 1 \
            -source hdfs \
            -method sequential
```

其中的选项-m指明了上一步构建的模型所在的目录。选项-method指明程序应该以串行模式运行，而不是使用Hadoop。像这样的小数据集，串行处理更好。而对于比较大的数据集，则必须采用并行操作，以保证运行时间控制在合理范围内。

测试程序运行完后，会输出下面这种信息。汇总信息中有正确或不正确分类文档的原始数量。

```
...
Summary
-------------------------------------------------------
Correctly Classified Instances          :       6398   84.9442%
Incorrectly Classified Instances        :       1134   15.0558%
Total Classified Instances              :       7532
=======================================================
```

我们只摘录了输出信息中的汇总部分，其他部分都省略了。从汇总信息中可以看出，朴素贝叶斯模型表现很好，正确率几乎达到了85%。除了总体数值（gross number），汇总信息中再没有任何与错误有关的详细信息。输出的下一部分是混淆矩阵（confusion matrix），其中有错误的详细信息。这部分输出如下：

```
Confusion Matrix
-------------------------------------------------------
a b c d e f g h i j k l m n o p q r s t <--Classified as
388                                       | 397 a = rec.sport.baseball
  386                                     | 396 b = sci.crypt
    396                                   | 399 c = rec.sport.hockey
      347                                 | 364 d = talk.politics.guns
        377                               | 398 e = soc.religion.christian
  12      304           18                | 393 f = sci.electronics
            281      14  43         21    | 394 g = comp.os.ms-windows.misc
              313          22 16          | 390 h = misc.forsale
     26 69       83 41                    | 251 i = talk.religion.misc
        45         225 13             11  | 319 j = alt.atheism
                     334            32    | 395 k = comp.windows.x
                       367                | 376 l = talk.politics.mideast
          19 23 15       307 13           | 392 m =comp.sys.ibm.pc.hardware
                          16 335          | 385 n = comp.sys.mac.hardware
                               371        | 394 o = sci.space
                                 393      | 398 p = rec.motorcycles
                                12 364    | 396 q = rec.autos
            11       22             305   | 389 r = comp.graphics
      102                             160 | 310 s = talk.politics.misc
                                        362 | 396 t = sci.med
Default Category: unknown: 20
```

实际输出和本书中的会稍有不同，因为此处为了适应纸面的宽度，我们把实际输出的长度压缩了。混淆矩阵中给出了所有正确和不正确分类的分解信息，因此可以看到模型在测试数据上犯了哪些错误。沿着对角线，我们可以看到大多数新闻组的分类做得都挺好，只有talk.religion.misc和talk.politics.misc两个新闻组的正确分类数量相对较少，显得比较突出。talk.politics.misc新闻组中几乎有1/3文档的分类不正确，被归到了talk.politics.guns中，即便不能说正确，但最起码表面上看起来是讲得通的。同样，talk.religion.misc中有69个文档被分到了soc.religion.christian中，这也能说得通。看到这些错误具有某种意义，我们应该更加安心了，因为它们表明模型是根据文档内容的实际含义来选择类别的。

有点讽刺的是，如果看到模型的性能不是出奇得好，心里倒会更加安心。比如说，如果在相同的训练数据上再次运行上面学到的模型，汇总信息将如下所示：

```
bin/mahout testclassifier -d 20news-train -m 20news-model\
   -type cbayes -ng 1 -source hdfs -method sequential
...
Correctly Classified Instances          :      11075   97.8876%
Incorrectly Classified Instances        :        239    2.1124%
Total Classified Instances              :      11314
```

对于这些训练中熟悉的数据，模型的正确率能够达到98%，对于这个具体问题来说，好得太过离谱。最好的机器学习研究人员在提到他们系统的正确率时，也只是说大概在84%到86%之间。我们在后续各章中还会针对模型评估问题展开更多的讨论，特别是探讨如何形成一种思路来判断模型真正应有的表现。

14.7　小结

这一章介绍了如何对20 Newsgroups数据集中的真实数据构建和训练分类器，并重点介绍了特征提取，以及如何为项目选择最佳算法。

无论从时间、精力，还是做好之后所能带来的回报来看，特征提取都是构建分类系统的重中之重。记住，对于不同的分类器而言，每个特征的价值并非总是一样的，你需要尝试多种组合来看它们的实际效果。原始数据是不能直接使用的。至于如何将原始数据变成可分类的形式，然后如何再将其变成Mahout学习算法所要求的向量，书中已经给出了详细介绍。

对于不同的学习算法而言，其作为输入变量的可接受值类型不同。14.5节专门概述了区分不同算法的各种差别，此外还介绍了一些其他的差异。现在，你对使用不同算法的优点和代价应该有个基本的认识，可以根据自己的理解选择适合自己项目的算法了。

最后，我们介绍了两种学习算法（SGD和朴素贝叶斯）在相同数据集（即20 Newsgroups数据）上的表现。

相信你对分类第一阶段（训练模型）已经有了深入的了解，我们接下来关注第二阶段，即对训练得到的模型进行评估和调优。这正是下一章的主题。

第 15 章

分类器评估及调优

本章内容

- 分类器评估中的基本问题
- 使用Mahout中的评估API
- 度量某个SGD分类器的性能
- 分类器中的常见问题及处理方法
- 分类器调优方法

由于在分类中处于重要地位，评估在Mahout被构建为一个基础构件。本章主要考虑分类过程的第二步即分类器的评估及调优过程，该过程为分类器的生产部署及性能维护做准备。我们会介绍如何在一个高层次上对分类器进行评估，并介绍利用Mahout API进行评估的细节。本章会给出一个如何使用Mahout API的例子。我们还会给出多个例子来强调如何利用Mahout评估API的性能指标和诊断功能来诊断分类器中的常见问题。本章最后会给出分类器调优策略和技术的一个讨论，涉及的范围从选择算法一直到调整学习率。

分类器评估也给出了一些陷阱，本章会给出一些方法来避免其中代价最大的陷阱。另一方面，由于分类器模型内部机理可能难以理解，所以分类器评估本身也存在挑战。

15.1 Mahout 中的分类器评估

要构建一个成功的分类器，评估是相当重要的一环。它涉及的远远不止是模型训练之后得到的表示成功程度的某个单一得分。实际上，它是从训练时就开始的一个反复迭代过程。构建分类器时早期的评估很有价值，这是因为它可以鼓励我们做出多种可能的尝试，包括选择不同的算法、配置和预测变量集合。除此之外，评估还可以对数据预处理流程中的错误进行校验，这些错误有可能会使分类器表现得比实际效果要好。

为评估分类器，Mahout提供了一系列的性能指标。这些指标主要包括正确百分比或称正确率（percent correct）、混淆矩阵（confusion matrix）、AUC和对数似然（log likelihood）等。朴素贝叶斯和补充朴素贝叶斯（complementary naive Bayes）最好用正确率和混淆矩阵来评估。对于SGD算法上述四种评估指标都可以，但是AUC和对数似然可能尤其有效，这是因为它们

能够深入到模型的置信度水平。

本节主要介绍三个方面的内容：首先是如何获得对模型性能的反馈，甚至在训练阶段就获得反馈；其次是如何解释Mahout提供的多个性能指标；最后是如何将损失代价集成到分类器评估当中。

15.1.1　获取即时反馈

Mahout中的评估与其他系统稍有不同。其中一个不同是Mahout能够在进行中提供即时的性能评估，它允许用户一次将目标变量分值和参照值输入给评估器来获得现场的性能反馈。这一点相当有用。与此形成对照的是，有些其他系统当中必须要批处理得到所有的得分和目标变量值之后才能进行评估。当需要记录学习系统的过程时，Mahout的这种在线评估能力就能提供方便。不仅如此，由于Mahout能够在系统运行时像在学习过程中一样对学习参数进行调整，因此上述做法也会带来实际价值。

上述获得进行中的指标的能力非常好，但是这些指标的含义到底如何?

15.1.2　确定分类“好”的含义

直觉上，大部分用户都希望分类器越精确越好，并且最好每次都将分类对象放入正确的类别当中。不幸的是，这种直觉并不能带来一个实际的评估机制。在现实当中，没有分类器会做到百分之百正确。实际上，一个很高的精确率往往表明测试中出现了错误，即看起来太好了不像是真的，那么很可能就不是真的！于是，到底如何度量分类器的“好”？一个有效的评估方法要取得成功，就必须要有实际真实的基准，这样才能认识到分类器的高效性。

诸如正确率之类的简单精度指标并不是对分类器进行评估和调优的最好指标。比如，考虑表15-1中两个假想的模型的对比。从中可以看到，如果采用一个简单的指标，一个几乎全错的分类器实际上会比某个偶尔还会对的分类器更有用。

表15-1　两个假想的分类器数据，该数据表明仅仅考察正确率会有一些缺陷。表中每列给出的是模型每种可能输出结果的频率得分值，每行给出了具体的正确值，最高得分都用加粗字体显示。其中，模型1几乎从不正确，但是仍然可能有用，而模型2却像一个“停摆的钟”

正确值	模型1			模型2		
	A	B	C	A	B	C
A	0.45	0.50	0.05	0.01	0.01	0.99
B	0.50	0.45	0.05	0.01	0.01	0.99
C	0.05	0.50	0.45	0.01	0.01	0.99

上面这个假想的例子表明，仅仅简单地看正确率往往不能展示模型的真正价值。上述表格中，每行对应一个样本，最左边是正确的答案。而每列代表的是三种可能输出结果中每一种的得分 。模型1从来都不对，但是它比模型2要有价值得多，尽管后者看上去精度明显要高一些。模型2采

用某个过分简单的固定规则来评分，有时会取得正确的结果，但这完全是纯随机的。而相比较而言，模型1却看上去始终能够从三种可能中选择错误的那个。

在很多应用中，由于模型2虽然有时会得到正确结果但却永远不会改变输出结果，所以模型1的结果可能在很多需求下会比模型2更好。对于上例而言，模型1的正确率是0，而模型2却为1/3。相比之下，模型1的平均对数似然是–0.8，而模型2为–3.5，后者更差一些。下一节当中，我们会讨论像对数似然这样的一些指标，它们能够更好地反映出我们想看到的性能好坏。

进一步地，对于模型来说，了解它何时可能正确和何时可能不对十分有用。有一些性能指标能够反映这种“自知之明”。上例中的模型1提供了这个信息，它对于两种选择表现得模棱两可，其中一种选择实际是正确的。而模型2看上去对于选择正确与否没有提供任何线索。当然，如果利用对数似然也能正确地显示出这种显著的差别。

然而，有时采用一个统一的性能指标并不是一件好事。当某些错误的代价远远高于其他错误时这一问题常常就会发生。

15.1.3 认识不同的错误代价

当某些错误的代价高于其他错误的时候，简单的精度指标的可用性会大打折扣。伪正例（false positive）可能就比伪反例（false negative）付出的代价要低很多。比如，本来没有癌症，但是误测出癌症了，可能需要付出重新检查和胆战心惊的代价，但是如果本来有癌症，但是没检测出来，则会付出生命的代价。

和上面的例子相比，一个不那么令人瞩目的反映漏报和误报的不同代价的例子是垃圾邮件过滤。如果垃圾邮件被判为正常邮件，用户可能只是抱怨，但如果正常邮件被判为垃圾邮件，用户可能会大为光火。每次将垃圾邮件误判为正常邮件只是浪费了用户的数秒时间，而将正常邮件标记为垃圾邮件带来的远远不止是不方便而已。如果这种情况相当频繁，那么用户有可能发现问题并采取相应行动。但是，如果上述错误十分罕见，用户就可能意识不到需要对这种可能出现的问题进行弥补，这样的话后果会不断升级。

Mahout分类器评估API的输出结果可以参考第13章和第14章给出的例子。这两章中给出的命令行工具可以给出评估的输出结果。某些情况下，常规的诊断方法已经足够，但是通常而言用户需要在自己的程序中使用API来了解运行的效果。下一节将详细介绍如何利用Mahout中的评估API来进行评估。

15.2 分类器评估 API

Mahout的分类器评估API包含了一系列的类，用于计算各种分类器性能指标。不论用户是否使用Mahout中的分类器，这些评估类或许都十分有用。

Mahout API类所支持的多种分类器指标如表15-2所示。每个指标都会在后续小节中详细介绍。其中包括如何在线和离线进行指标的计算。在线学习算法也提供了API，用于在训练阶段访

问上述的几个性能指标。正如在表15-2中可能看到的那样，所有的指标的支持方式并不一致。表15-3中按照类分别列出了相关功能。

表15-2　Mahout通过多个API所支持的分类器性能指标

指　标	支持的类
正确率	CrossFoldLearner
混淆矩阵	ConfusionMatrix, Auc
熵矩阵	Auc
AUC	Auc, OnlineAuc, CrossFoldLearner, AdaptiveLogisticRegression
对数似然	CrossFoldLearner

表15-3　支持分类器性能评估的类

类	方　法
Auc	auc()、confusion()、entropy()
OnlineAuc[①]	auc()
OnlineSummarizer	
ConfusionMatrix	
AbstractVectorClassifier	loglikelihood(int, vector)
CrossFoldLearner	auc()、percentCorrect()、loglikelihood()
AdaptiveLogisticRegression	auc()

需要注意的是，上述各类的使用方式有所不同。上面的学习算法，包括AdaptiveLogisticRegression、CrossFoldLearner以及AbstractVectorClassifier都提供了对所学模型进行评估的方法。而与此形成对照的是，Auc和OnlineAuc是相互独立的类，它们在给定得分和目标参照值之后计算性能指标。而最后，OnlineSummarizer对任意指标计算总的统计信息。

下面我们将介绍计算这些指标的过程细节。

15.2.1　计算AUC

当所要评估的模型的目标变量为二值且该模型的输出得分为连续值时，采用AUC指标十分有用。例如，某个分类器计算一个物品是否包含某个属性的0到1之间的概率，该分类器就适合用AUC来计算。AUC并不要求分类器的输出得分一定是概率估计值，输出值范围差异很大的分类器都可以利用AUC来比较。

读者不必深入理解AUC的意义，只需要记得AUC是指一个随机选择的yes实例要比一个随机选择的no实例的得分要高的概率。一个输出的得分和目标变量毫不相干的分类器的AUC值在0.5左右，而一个完美模型的AUC得分为1。AUC得分在0.7到0.9之间的模型通常被认为“好”。对于欺诈检测和点击分析来说，上述区间也是模型产生有用的精确结果的区间范围。当模型的预测结果正好和目标变量相反时，AUC的得分为0。

① 原文该表格不全，故根据上下文进行了修改。另外，onlineAuc实际上是一个接口，不是类。——译者注


No. 15

不论出于实际还是理论上的考虑，AUC最好采用如下计算方式，即从所有数据中保留一部分作为测试数据，然后在这些已知目标结果的测试数据上比较分类器。不幸的是，AUC通常仅适合于计算输出得分值的二值分类模型而不是硬分类结果。当然，现在也存在AUC的一些扩展能够支持非二值的分类结果，但是目前Mahout并不支持这些扩展。

Mahout中，如果二值分类器在测试数据上的输出得分值和标准目标变量值已知的话，存在多种计算AUC的方法，其中包括`org.apache.mahout.classifier.evaluation`包的`Auc`类。此外，还存在`OnlineAuc`的接口实现以及`org.apache.mahout.math.stats`包中的`GlobalOnlineAuc`或`GroupedOnlineAuc`类。对于上述提到的所有类，只要给出测试数据上的真实目标变量值和分类器的输出得分值就可以计算AUC。而对于这些类，AUC的值均来自`auc()`方法。

`OnlineAuc`与`Auc`有所不同，在数据载入过程中的任意时刻`OnlineAuc`都可以给出AUC的估计值，尽管每个元素插入的开销稍大，但这种估计所需要的开销却很小。尽管`Auc`和`OnlineAuc`中的算法的结果基本相同，但是由于涉及对数千的样本排序，`Auc.auc()`的开销比`OnlineAuc.auc()`更大。

下面的代码清单给出了上述两个类当中如何从文件中读入分类器得分和目标变量值然后计算AUC的过程。

代码清单15-1 向AUC指标类传递数据

```
Auc x1 = new Auc();
OnlineAuc x2 = new GlobalOnlineAuc();
BufferedReader in = new BufferedReader(new FileReader(inputFile));
int lineCount = 0;
String line = in.readLine();
while (line != null) {
  lineCount++;
  String[] pieces = line.split(",");
  double score = Double.parseDouble(pieces[0]);
  int target = Integer.parseInt(pieces[1]);
  x1.add(target, score);
  x2.addSample(target, score);
  if (lineCount%500 == 0) {
    System.out.printf("%10d\t%10.3f\t%10d\t%.3f\n",            ❶ 获得输入后x2计算AUC
                 lineCount, score, target, x2.auc());
  }
  line = in.readLine();
}
System.out.printf("%d lines read\n", lineCount);                 ❷ x1和x2的结果
System.out.printf("%10.2f = batch estimate\n", x1.auc())           实际上完全一样
System.out.printf("%10.2f = on-line estimate\n", x2.auc());
```

上述程序当中，x1和x2计算出的AUC值几乎完全一样，但是对于x2来说，在扫描数据的过程中，它就会产生AUC在过程中的估计值❶。最后，x1和x2得到的结果本质上相同，但是它们的实现方式有所不同❷。

在CrossFoldLearner和AdaptiveLogisticRegression类中auc()及getLogLikelihood()方法的训练过程中，可以计算最新的AUC或对数似然值。通过这些值可以对训练的多个模型进行比较。这个过程通过AdaptiveLogisticRegression来实现，其中它可以在训练中对训练参数进行调整。

尽管AUC通常是二值目标变量及输出分值时的黄金标准，但当目标变量不止两类或者不会输出得分值时，还需要寻找其他好的评估指标。

15.2.2　计算混淆矩阵和熵矩阵

对于输出非分值结果的分类器来说，可能最直接的指标就是混淆矩阵。混淆矩阵是模型输出结果和已知正确目标值的交叉表。矩阵的每一行对应真实目标值而每一列对应模型的输出值。矩阵第i行j列的元素值为类别i的测试样本被模型分到类别j中的数目。一个好模型对应的混淆矩阵的大元素都集中在对角线。这些对角线元素指的是类别i中样本被正确分到i类中的数目。

下面给出了一个混淆矩阵的例子。

```
=======================================================
Confusion Matrix
-------------------------------------------------------
a    b    c    d    e    f    <--Classified as
9    0    1    0    0    0     |  10       a      = one
0    9    0    0    1    0     |  10       b      = two
0    0    10   0    0    0     |  10       c      = three
0    0    1    8    1    0     |  10       d      = four
1    1    0    0    7    1     |  10       e      = five
0    0    0    0    1    9     |  10       f      = six
Default Category: one: 6
```

这个例子中的数据是我们自己造的，该例子给出了一个典型的精确分类器，我们可以看到，矩阵中的大元素都集中在对角线上。

代码清单15-2给出了利用org.apache.mahout.classifier包中的ConfusionMatrix类来计算混淆矩阵的过程，该过程使用了训练集或测试集中真实的类别结果，以及分类器的输出结果。这段代码会对数据进行两次扫描：一次是获得所有的目标变量列表，另一次是填充混淆矩阵。通常情况下第一次扫描并不一定要做，这是因为所有的变量值可以通过其他方法来得到。程序中的关键点就是你需要传入已知的正确目标变量值和模型计算出的目标变量值。

代码清单15-2　构建混淆矩阵

```
BufferedReader in = new
     BufferedReader(new FileReader(inputFile));         <-- 读入并保存值
List<String> symbols = new ArrayList<String>();
String line = in.readLine();
while (line != null) {
  String[] pieces = line.split(",");
  if (!symbols.contains(pieces[0])) {
```

```
    symbols.add(pieces[0]);
  }
  line = in.readLine();
}

ConfusionMatrix x2 = new ConfusionMatrix(symbols, "unknown");

in = new BufferedReader(new FileReader(inputFile));      <—— 对真实值/预测值对计数
line = in.readLine();
while (line != null) {
  String[] pieces = line.split(",");                     <┐ 输入中包括目标值和模型输出值
  String trueValue = pieces[0];                           |
  String estimatedValue = pieces[1];                     
  x2.addInstance(trueValue, estimatedValue);             <┐ x2得到真实结果和得分值
  line = in.readLine();                                   |
}
System.out.printf("%s\n\n", x2.toString());
```

ConfusionMatrix产生的混淆矩阵取决于待比较的分类器的输出结果，或者在二值分类下也取决于所选择的阈值。基于混淆矩阵的分类器要求模型输出单一结果（即属于哪个类），因为此时只有最后的结果才有作用，所以这时会丢失模型对输出结果的确信度。当分类器能输出概率得分值时，可以通过计算一个称为熵矩阵的混淆矩阵变体来避免上述的信息丢失。

与混淆矩阵一样，熵矩阵中的每行对应的是实际的目标值，每列对应模型的输出值。区别在于，熵矩阵的元素计算的是每个真实或估计类别组合的概率的平均对数值。这里之所以使用平均对数来计算主要是受到概率分布近似值最优求解的数学原理的启发。该平均对数概率表达的是，通过比较分到某个特定类别的分类器似然和该类别全面性的普遍程度来得到某个特定的分类器结果的非正常程度。一个好的模型在对角线上会得到绝对值较小的负数，而非对角线上为绝对值较大的负数。这可以类比与混淆矩阵对角线上的较大值和非对角线上的较小值。

熵矩阵和对数似然之间有着十分有用的关联，即熵矩阵对角线元素的加权平均值就是对数似然。对数似然通常通过计算在目标类别上得分的平均对数值来得到。下一节当中将会进一步深入讨论对数似然。

Auc同时可以用于计算混淆矩阵和熵矩阵，但是目前它仅仅支持二值的目标变量值。读者可以采用如下方法往Auc对象中加入数据并打印出混淆矩阵和熵矩阵：

```
Matrix entropy = x1.entropy();
for (int i = 0; i < entropy.rowSize(); i++) {
  for (int j = 0; j < entropy.columnSize(); j++) {
    System.out.printf("%10.2f ", entropy.get(i, j));
  }
  System.out.printf("\n");
}
```


目前为止，Auc类仅仅适用于二值目标变量的情况。对Auc类进行扩展以支持更多的变量类型或许是Mahout未来版本的一个功能。

当模型输出概率得分值时，正确答案得分的概率值能够很好地度量模型的好坏程度。如果该值来自大规模存留样本的平均，那么它能提供模型质量的一个十分有用的单一度量指标。该指标

称为平均对数似然。尽管有误解的可能，但上述指标往往仍被简称为对数似然。

15.2.3　计算平均对数似然

对数似然是基于对输出结果的确信或非确信程度来对模型进行评分的方法。对数似然要求模型输出一个概率得分结果。如果模型对错误结果给出很低的概率而对正确结果给出较高的概率那么模型就获得信用值。对数似然评估方法的一个非常有用的特点是，即使模型并没有精确地得到正确结果，但是在可能结果的范围有所限制时模型仍然得到部分的信用值。

对数似然具有一个令人满意的数学性质，即如果所用模型能够精确复制目标变量的真实概率分布，那么只能最大化对数似然得分。对数似然和AUC可以互相补充。对数似然可以用于多值输出的情况，而AUC仅能用于二值输出。另一方面，AUC可以接受任意类型的得分，而对数似然要求得分反映概率的估计值。AUC的值要求在0到1之间以便对模型进行比较，而对数似然没有限定取值范围。

由于不同样本之间的对数似然相差很大，因此单个训练样本的对数似然值不是特别有用，但是多个样本上的平均值很有用。读者可以利用OnlineSummarizer来计算分类器的平均对数似然估计值，然后汇总出这些估计值的中位数、平均数和上下四分位数等统计量。具体做法如下：

```
OnlineSummarizer summarizeLogLikelihood = new OnlineSummarizer();
for (int i = 0; i < 1000; i++) {
  Example x = Example.readExample();
  double ll = classifier.logLikelihood(x.target, x.features);
  summarizeLogLikelihood.add(ll);
}

System.out.printf(
    "Average log-likelihood = %.2f (%.2f, %.2f) (25%%-ile,75%%-ile)\n",
    summarizeLogLikelihood.getMean(),
    summarizeLogLikelihood.getQuartile(1),
    summarizeLogLikelihood.getQuartile(2));
```

上述代码首先读入样本然后计算分类器的对数似然，最后将结果传递给OnlineSummarizer。读入相关信息之后，可以利用getMean()和getQuartile()等方法来获得整个对数似然分布的统计信息。

对数似然的最大值为0，而最小负值却没有限制。对于精度非常高的分类器来说，平均对数似然的值应该接近于分类器的平均正确率和目标类别数目的乘积。而对于精度稍差的分类器，特别当目标类别很多的情况下，平均正确率倾向于0，此时分类器的比较相当困难。另一方面，对数似然能够在分类器很差即分类器很少甚至从来没有分对过，导致平均正确率为0的情况下，仍然能够区分分类器的好坏。


No. 16

对于内部开发者而言，对数似然或许是一个不错的模型比较方法，但是当要向非技术人士展示结果时，采用正确率甚至混淆矩阵可能效果更好。

诸如AUC和对数似然之类的指标能够告诉我们模型的总体性能优劣，但是却无法给出模型优劣的原因。为了解这个原因，需要进入模型内部对其进行深入剖析。

15.2.4 模型剖析

对模型进行深入剖析是了解哪些特征导致结果出现较大差异的途径之一。Mahout可以使用org.apache.mahout.classifier.sgd 包中的 ModelDessector 来对 AbstractVectorClassifier的继承类进行剖析处理。

我们还记得，作为预测变量的特征必须要进行词条化和向量化处理才能在学习算法中使用。这些特征向量属于模型理解所需要的考察线索之一。ModelDissector以一个特征向量、该特征向量的构建轨迹和模型作为输入值。然后对特征向量进行微调来观察结果的变化情况。通过在一系列样本上求平均，可以确定不同特征的效果。当需要输出汇总结果时，ModelDissector会返回那些变量及其值的列表，这个列表给出了最大的平均效果。

接下来的代码片段给出的是如何利用ModelDissector来给出对模型输出结果具有最大影响的变量列表的过程。第一步是关闭模型使得所有的学习和正则化过程结束，第二步是建立ModelDissector及其关联轨迹词典。

```
model.close();

Map<String, Set<Integer>> traceDictionary =
    new HashMap<String, Set<Integer>>();
ModelDissector md = new ModelDissector();

encoder.setTraceDictionary(traceDictionary);
```

一旦预备条件就绪的话，就需要在多个样例上反复循环。对于每个样例，需要消除轨迹，模型剖析器要随着特征向量、轨迹和模型而更新。

```
for (... lots of examples ...) {
  traceDictionary.clear();
  Vector v = encodeFeatureVector(example, actual, leakType);
  md.update(v, traceDictionary, model);
}
```

最后，可以要求模型剖析器输出模型的特征名称及其权重的汇总结果列表。

```
List<ModelDissector.Weight> weights = md.summary(100);
for (ModelDissector.Weight w : weights) {
  System.out.printf("%s\t%.1f\n",
          w.getFeature(), w.getWeight());
}
```

这里的循环语句是伪代码但其余代码都来自实际工作的代码。代码的关键因素在于编码当中使用了轨迹词典，该词典记录的用于后面模型剖析的线索信息。用户不需要将很多训练样例放入编码器中来获得大量的线索信息。一般而言，如果有几千或者几万样例，基本上应该就可以了。

ModelDissector的输出结果对于一个表现很差的模型的诊断很有帮助。在一个训练恰当的模型中，最重要的变量及其值应该能被该问题的领域专家所理解。但在一个失败模型当中更常见的是，最重要的变量明显很荒谬。在接下来的几节当中，读者会看到成功和失败的模型案例。

15.2.5　20 Newsgroups语料上SGD分类器的性能指标计算

14.4节利用SGD算法写了一个分类器来对20 Newsgroup语料分类。本节将对上述例子进行扩展以集中关注分类器的性能评估。这里我们会对整个学习程序连同性能指标一起进行剖析，以便了解这些部分如何一起工作。下几节中，我们会在该例子中注入几个典型的漏洞（bug），来观察它们对结果的影响。

下面的代码清单给出了在20 Newsgroups语料上训练出一个`AdaptiveLogisticRegression`模型的过程。这个例子与14.4节中的例子有些类似。

代码清单15-3　一个完整的带过程诊断的模型训练程序

```
private static final Analyzer analyzer =
    new StandardAnalyzer(Version.LUCENE_30);
private static final FeatureVectorEncoder encoder =
    new StaticWordValueEncoder("body");
private static final FeatureVectorEncoder bias =
    new ConstantValueEncoder("Intercept");

public static void main(String[] args) throws IOException {
  File base = new File(args[0]);
  int leakType = 0;
  if (args.length > 1) {
    leakType = Integer.parseInt(args[1]);
  }
  Dictionary newsGroups = new Dictionary();
  encoder.setProbes(2);

  AdaptiveLogisticRegression learningAlgorithm =
      new AdaptiveLogisticRegression(20, FEATURES, new L1());
  learningAlgorithm.setInterval(800);                        ❶ 设定较短的生成时间
  learningAlgorithm.setAveragingWindow(500);                 ❷ 为平均值设定小窗口

  List<File> files = Lists.newArrayList();
   File[] directories = base.listFiles();
       Arrays.sort(directories, Ordering.usingToString());
  for (File newsgroup : directories) {
    if (newsgroup.isDirectory()) {
      newsGroups.intern(newsgroup.getName());
      files.addAll(Arrays.asList(newsgroup.listFiles()));
    }
}
  Collections.shuffle(files);                                ❸ 随机化训练次序

  int k = 0;
  for (File file : files) {
    String ng = file.getParentFile().getName();
    int actual = newsGroups.intern(ng);

      Vector v = encodeFeatureVector(file, actual, leakType);
    learningAlgorithm.train(actual, v);
    k++;

    State<AdaptiveLogisticRegression.Wrapper, CrossFoldLearner> best =
```

```
            learningAlgorithm.getBest();

      if (best != null && k % 500) {
        CrossFoldLearner model =
            best.getPayload().getLearner();
        double averageCorrect = model.percentCorrect();
        double averageLL = model.logLikelihood();
        System.out.printf("%d\t%.3f\t%.2f\t%s\n",
              k, averageLL, averageCorrect * 100, leakLabels[leakType % 3]);
      }
    }
    dissect(newsGroups, learningAlgorithm, files);
  }
```

❹ 剖析最终的最优模型

上面的程序一开始对学习算法进行分配和配置处理。这里为了简单起见，选择的是`AdaptiveLogisticRegression`算法，使用该算法不需要担心学习参数。如果不介意学习参数微调的话，读者也可以选择`CrossFoldLearner`算法。上述程序为内部的进化算法设置了一个很短的生成时间，这是因为数据集相对较小算法可以学习得很快❶。类似地，诊断用的平均窗口大小也设置得很短，以便能够快速显示学习算法的运行状况❷。

在准备训练过程时，需要一个训练集中所有文档的列表。我们也有可能想对这个列表进行随机排序以提高学习的效率。为了训练模型，程序在随机排序的文档列表上反复循环迭代❸。整个训练过程包括到向量形式的转换以及模型的实际训练。一旦`AdaptiveLogisticRegression`中的进化算法开始运行，它将提供模型池中最好的模型。可以利用该模型来获得近期训练样本的平均对数似然和正确率。

当完成训练之后，可以对模型进行剖析❹。具体的剖析过程在如下代码清单中给出。

代码清单15-4 20 Newsgroups语料上的模型剖析

```
private static void dissect(Dictionary newsGroups,
          AdaptiveLogisticRegression learningAlgorithm,
          Iterable<File> files) throws IOException {
  CrossFoldLearner model =
        learningAlgorithm.getBest().getPayload().getLearner();
  model.close();

  ModelDissector md = new ModelDissector();
  Map<String, Set<Integer>> traceDictionary =
          Maps.newTreeMap();
  encoder.setTraceDictionary(traceDictionary);
  bias.setTraceDictionary(traceDictionary);

  for (File file : permute(files, rand).subList(0, 500)) {
    String ng = file.getParentFile().getName();
    int actual = newsGroups.intern(ng);

    traceDictionary.clear();
    Vector v = encodeFeatureVector(file, actual, leakType);
    md.update(v, traceDictionary, model);
  }

  List<String> ngNames = Lists.newArrayList(newsGroups.values());
  for (ModelDissector.Weight w : md.summary(100)) {
```

❶ 关闭模型以确定权重

❷ 提供轨迹词典

❸ 清除词典以防止内存阻塞

```
    System.out.printf("%s\t%.1f\t%s\t%s\n", w.getFeature(), w.getWeight(),
        ngNames.get(w.getMaxImpact() + 1));
  }
}
```

模型剖析时，首先必须确认所有的模型参数都已经确定❶。关闭一个在线模型并不能防止进一步的训练，它能够确认当前已经拥有剖析的一致状态。

剖析中，需要将特征编码到一个称为轨迹词典的对象中❷。该对象记录的是哪个预测变量和值分配到特征向量中的哪个位置。这样做会大大降低编码的速度，因此并非常规的做法。通过清除每个或每几个样本的轨迹词典，可以避免轨迹词典在内存中占据太大空间❸。在对每个样本编码并且记录特征提取的细节之后，就将细节信息传给模型剖析器。

一旦处理完一定量的训练样本之后，就可以要求模型剖析器返回最重要的变量及其值列表。本例当中返回的是前100个。

如果运行这个例子，我们会发现正确率会如图15-1所示那样变化。

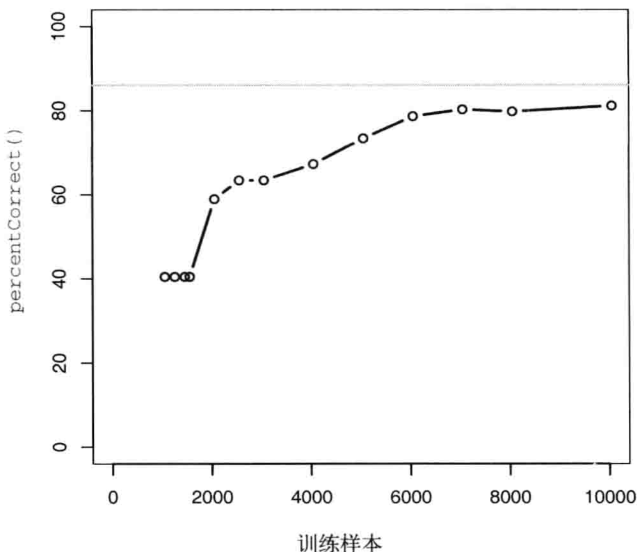


图15-1　平均正确率随训练样本数增加而增长的示意图。上面的灰色横线给出了预期的最大结果，该结果来自现有相关工作中所报告的最优结果

如果对图15-1的数据进行多篇扫描，实际的性能可能会在中间80%那段接近当前最优水平。对数据进行多遍扫描不会像先前在未知的额外数据上训练时那样使模型有大幅提升，这是因为同一数据的多遍扫描没有带来训练数据多样性的变化。幸运的是，基于Mahout的分类当中永远都不必担心样本太小，这是因为Mahout之所以出名的最大原因就是能够处理大数据集。当然对数据进行多遍扫描仍然可以提高模型的性能，因为每一遍扫描系统一定程度上都在学习。

下面给出了一个模型剖析后的典型运行结果，该结果与图15-1有点类似。下面各列分别是特征名称（这里用的是单个词语）、该特征的最大权重和在该权重下的目标变量值。

```
body=space        2.1  sci.space
body=sale         1.9  misc.forsale
body=car          1.9  rec.autos
body=windows      1.8  comp.os.ms-windows.misc
body=mac          1.7  comp.sys.mac.hardware
body=bike         1.7  rec.motorcycles
body=apple        1.5  comp.sys.mac.hardware
body=gun          1.5  talk.politics.guns
body=baseball     1.5  rec.sport.baseball
body=graphics     1.5  comp.graphics
body=key          1.5  sci.crypt
body=x            1.4  comp.windows.x
body=dod          1.4  rec.motorcycles
body=orbit        1.4  sci.space
body=clipper      1.4  sci.crypt
body=encryption   1.3  sci.crypt
body=window       1.3  comp.windows.x
body=image        1.2  comp.graphics
body=hockey       1.2  rec.sport.hockey
body=atheists     1.2  alt.atheism
```

这些特征显然相当合理，只有单词dod可能有点意外。该单词显然是rec.motorcycles Usenet讨论组的一个特征。考察训练文本会发现一个有趣的现象，有一篇长贴，它反复将dod作为无意义词使用。如果这个词的生命周期很短暂的话，该特征可能对于将来的性能没有什么帮助，但至少也不会有什么坏处。

迄今为止，我们已经了解了如何衡量分类器"好"的程度。留给我们做的是理解这些指标的含义。接下来我们介绍在性能指标计算中可能出现的典型问题以及如何诊断并解决这些问题。

15.3 分类器性能下降时的处理

使用实际数据和实际分类器时，几乎可以肯定的是，初始的建模尝试会失败，有时甚至会很惨烈。与通常的软件工程不同，模型的失败通常并不像废弃的空指针或内存溢出异常那么明显。与此相反，一个失败的模型可能会输出异常精确的结果。这样的模型同时也会输出错误率很高的结果，看上去模型不太对头。如果发现，特别在模型构建的初期，分类器的结果极端精确或者糟糕，那么这一结果应该多少值得怀疑，这一点相当重要。

本节将利用前面介绍过的技术，帮助读者理解性能指标如何帮助了解在分类器生命周期中的各种"磕磕碰碰"。为实现这一点，读者必须了解分类器中的常见问题以及这些问题如何通过性能指标来体现。

分类器模型构建过程中的两个最普遍的问题称为目标泄漏（target leak）和特征提取崩溃（broken feature extraction）。当训练数据的某个特征在某种程度上是目标变量提供的信息在生产环境中并不出现的，那么就称为发生了一次目标泄漏。此时的分类器就相当于自己出题考试然后提交解答一样，不出意外，这种情况下分类器可以取得很好的效果。这类问题的出现可能相当隐蔽，使其很难被发现。如果不出现灾难性的结果，很差的特征提取也很难检测出来。下面几节将考察上述两类问题的案例。

15.3.1 目标泄漏

目标泄漏是由于目标变量的信息包含在用于训练分类器的预测变量中而带来的漏洞。目标泄漏的一个最主要的征兆是结果的性能好得难以置信。

为帮助读者理解如何避免目标泄漏，我们可以在代码清单15-3的20 Newsgroups分类代码中有意地插入一个目标泄漏。具体地，我们可以对程序进行简单的修改，在训练样本中增加一个目标泄漏：

```
private static final SimpleDateFormat("MMM-yyyy");
// 1997-01-15 00:01:00 GMT
private static final long DATE_REFERENCE = 853286460;

...
  long date = (long) (1000 *
      (DATE_REFERENCE + target * MONTH + 1 * WEEK * rand.nextDouble()));
  Reader dateString = new StringReader(df.format(new Date(date)));
  countWords(analyzer, words, dateString);
```

和以前不一样的是，这里的代码在数据的日期字段注入了一个目标泄漏。选择日期字段可以使得来自同一新闻组（Newsgroup）的文档看上去都出自同一月份，而不同新闻组（Newsgroup）的文档出自不同月份。日期、时间和ID号码都是常见的目标泄漏源，但我们可以选择其中的任意一个。甚至是目标变量本身都可能偶然地包含在预测因子中。正如我们所料的那样，这种错误是一切目标泄漏的根源。

当训练带有上述目标泄漏的模型时，模型的精度会变得相当好。图15-2给出了此时模型的运行结果。可以看到，当目标泄漏仅和主题行一起出现时，精度可以达到100%。这么高不太现实，而目前相关研究工作中报道的最高值大概在86%左右。当把目标泄漏加到从标题行和消息体的所有文本中时，学习算法会运行更长的时间，但是不久仍然会获得一个不可思议的高精度。

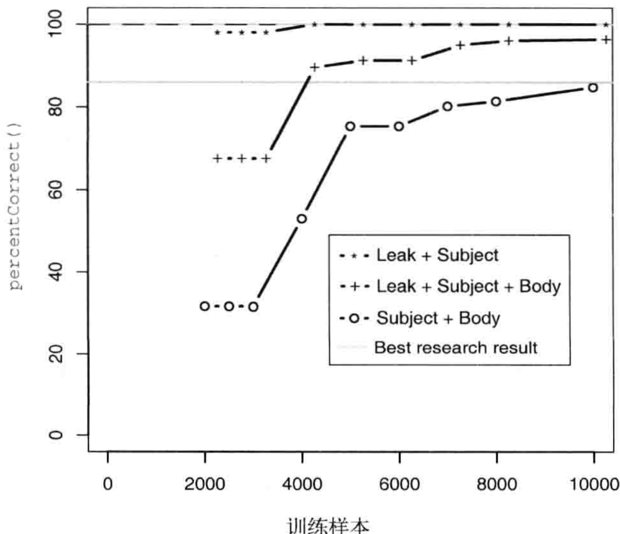


图15-2　目标泄漏导致的不可思议的高精度值，但并不总是很快就达到高精度

从图15-2的结果可以看出，代表包含目标泄漏的两条线很快都能达到100%的精度。对于这个问题来说，这种极端的精度不可信。另一方面，不带漏洞的SGD模型只能刚刚达到该问题在相关工作中的最优值。此时的性能水平不仅很好，而且可信。

当面对实际数据而不是大家认真研究过的研究数据时，不可能从一开始就知道“过好”到底是多好。如果模型表现得异常好，或者如果在新数据的表现显著差于交叉检验的预测结果，那么就值得观察一下`ModelDissector`的输出结果。

下面给出的是对上述某个存在目标泄漏的例子进行模型剖析的前些行的结果。其中每一列分别达标词项、权重和权重对应的新闻组。

```
body=feb-1998           3.7   sci.electronics
body=dec-1997           3.6   comp.graphics
body=sep-1997           3.5   sci.med
body=mar-1998           3.3   comp.sys.mac.hardware
body=jul-1997           3.3   talk.religion.misc
body=jun-1997           3.2   misc.forsale
body=apr-1998           3.2   rec.motorcycles
body=jul-1998           3.1   talk.politics.misc
body=jun-1998           3.0   comp.os.ms-windows.misc
body=mar-1997           2.9   sci.space
body=apr-1997           2.9   talk.politics.guns
body=jan-1998           2.8   soc.religion.christian
body=nov-1997           2.8   comp.sys.ibm.pc.hardware
body=may-1997           2.8   comp.windows.x
body=feb-1997           2.7   rec.sport.baseball
body=aug-1998           2.6   alt.atheism
body=may-1998           2.6   rec.autos
body=oct-1997           2.5   rec.sport.hockey
body=gun                2.1   talk.politics.guns
body=aug-1997           2.1   sci.crypt
body=windows            2.0   comp.os.ms-windows.misc
body=consciously        1.8   sci.electronics
body=taber              1.8   sci.electronics
body=uw.pc.general      1.8   sci.electronics
body=hearing            1.8   sci.med
body=car                1.7   rec.autos
body=sdennis@osf.org    1.7   misc.forsale
body=1024x768x24        1.7   comp.graphics
body=contradicts        1.7   talk.religion.misc
body=market             1.7   comp.graphics
body=passes             1.7   misc.forsale
body=att-14,796         1.7   talk.religion.misc
body=bars               1.7   sci.med
```

从上面的结果可以看出，权重最大的那些特征基本都是日期。除了标志大规模季节变化的日期特征如圣诞节之外，其他日期特征几乎都不太可能很有效地预测主题。而这里几乎所有的前20个特征都是特定日期，这完全无法令人相信，因此唯一的合理结论就是这些特征代表了某个目标泄漏。当然，这里我们有意地注入了一个目标泄漏，因此找到一个漏洞完全不会令人意外。

另一方面，我们也可以通过细微考察排名靠前的非漏洞特征来发现问题。`AdaptiveLogistic`

Regression通过调节学习率来对性能进行优化。在存在目标泄漏的情况下，该算法的一个好处就是一旦发现漏洞特征就会很快降低训练学习率。同样，其他无关特征的权重就会被冻结而不会像正常模型为获得好的性能那样将权重降低到0。因此，我们会发现taber、market、passes及bars等词都分别被列为sci.electronics、comp.graphics、misc.forsale和sci.med等类别的高权重特征。这些特征在最终模型中的存在也表明学习率在没有正当调整之前被冻结。

15.3.2　特征提取崩溃

另一个常见的问题是特征提取部分在某种程度上发生崩溃。和目标泄漏导致性能异常好不同，特征提取崩溃会导致性能远低于预想值。

下面的代码给出的是Lucene词条化工具如何对20 Newsgroups数据进行词条化处理的过程：

```
TokenStream ts = analyzer.tokenStream("text", in);
ts.addAttribute(TermAttribute.class);
while (ts.incrementToken()) {
  String s = ts.getAttribute(TermAttribute.class).term();
  words.add(s);
}
```

一个很容易犯的错误就是使用s = ts.toString()而不是上述正确的term()调用来获取文本词条，尽管后者有些晦涩。如果犯了这个错误的话，那么词条化之后在词条中就会保留额外的信息从而导致在不同位置上的相同词语却看上去像两个词。这使得预测变量受到影响，从这些变量来学习，就算不是不可能也会十分困难。

图15-3给出了当存在词条化错误时性能上可能表现出的结果。正确率从没比5%高多少，当然也就是说模型的结果可能是随机选择的结果。

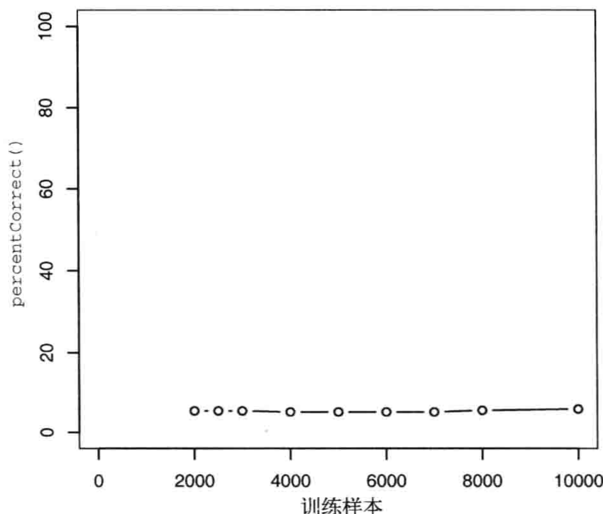


图15-3　存在词条化错误的系统性能从不超过某个随机选择系统的预想水平。灰线给出的是随机选择情况下的预计性能

由于所有输入数据都因词条化错误而崩溃，模型没有信息可用于产生好的结果，所以上述例子十分极端。实际中，更普遍的情况是某些字段崩溃而其他的都还好。如果崩溃字段对模型有帮助，那么这种部分的崩溃情况会导致性能的降低，但是此时的性能曲线不一定就像图15-3那样平坦，这是因为未崩溃的字段仍然承载了一定的信息。

发现上述问题的最好方法是计算系统中值的汇总统计信息。如果值是连续的，就利用`OnlineSummarizer`来检查均值、最小值、最大值和四分位数。这些数值应该看上去合理。对类别型、单词型和文本型数据，我们应该计算出词条的数目并在对数-对数坐标轴上画出数目和排名之间的关系图。对于20 Newsgroups数据来说，我们可能会得到类似图15-4的结果。对于单词型和文本型数据，对每个字段可能会出现类似的结果。而在上述注入词条化错误的情况下，基本上所有的词项频率都正好是1。

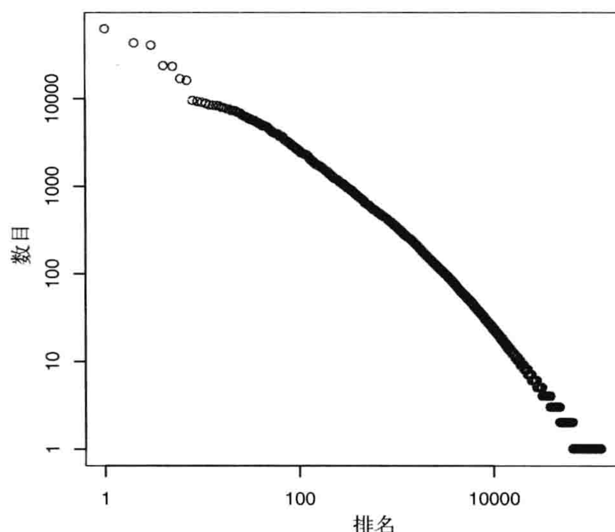


图15-4 在20 Newsgroups数据上一个正确词条化工具处理下的词条数目与排名之间的关系图。在词条化工具崩溃时，可能所有的词项数目都是1

表15-4给出了一些当加入`insertTokenError`时产生的错误词条，同时也给出了相应的正确词条。记住这里的所有的括号、逗号和数字都是错误词条的一部分。

表15-4 错误词条导致特征提取崩溃的例子

错误词条	正确词条
(term=elastic,startOffset=1705,endOffset=1712,...)	elastic
(term=elastic,startOffset=2064,endOffset=2071,...)	elastic
(term=failed,startOffset=1236,endOffset=1242,...)	failed
(term=failed,startOffset=245,endOffset=251,...)	failed
(term=failed,startOffset=974,endOffset=980,...)	failed

从表15-4可以看出，如果因为疏忽对于同一词条保留了太多的信息会使得该词条重复多次，而且每次看上去是不同的词条。这种情况会使得这些错误词条对于分类来说几乎没用，这是因为在另外一篇文档中出现在同一位置几乎不太可能。

到现在为止，我们知道诸如目标泄漏和崩溃的特征提取之类的问题都会使得分类器失效，下面我们转向一个更乐观的话题，即如何利用评估来调整和优化分类器以提高其性能。

15.4　分类器性能调优

除了纠正训练数据上的错误之外，还有很多方法可以提高分类器的性能。这些方法包括通过改变问题本身来使分类器分类更容易。其他方法还包括使得分类器在给定数据上运行性能更优。第一种情况下，有可能可以找到新的预测变量、预测变量的组合或转换结果来帮助提高分类器。另一种可能是去除那些对分类没有帮助的预测变量或者对目标变量进行简化。第二种情况下，可以在学习算法中调整学习率或者特征编码策略。

任一情况下，当对分类器进行调整时，一定要对修改前后的性能进行仔细和真实的度量。如果所做的修改导致的性能差异不是特别显著的话，只检查一些常规的样本往往并不够。实际上，一旦系统在合理运行，大部分调整都只会导致性能的轻微提高或降低，而这些性能的变化只能通过精确的测量才能发现。

15.4.1　问题调整

一提到分类系统的调整，大部分人都会马上想到调整学习算法以尽可能地提高精确度。然而，这种想法可能会因为错过最佳的机会而带来麻烦。通常而言，最大的提高来自对问题本身的调整。这包括输入、输出的改变甚至对输出解释的改变。

1. 剔除无价值的变量

有些变量对于分类器帮助不大甚至毫无作用，因此将这些变量从分类器中剔除可能会更好。有点看似反常的是，去除这些信息之后由于可以使得学习算法集中在更少的变量上，因此可以在有效训练时间内学到更好的模型来提高分类精度。15.4节中给出的目标泄漏的例子就是这个现象的很好的一个例子。在该例当中，当目标泄漏周围有很多文本时，学习算法花了很多的时间来学习这些并不重要的目标泄漏模型。反之，也可以说这些漏洞变量自己分散了学习算法的注意力，阻碍它去关注那些在主题和文本主体部分真正有价值的变量。

正则化是上述思路的一个更一般的版本。正则化是通过降低某些变量的影响甚至完全剔除这些变量来避免过拟合的一种数学技术。它通过同时最小化学习算法的错误和模型复杂度来实现。在Mahout的术语中，对复杂度进行惩罚的数学函数称为先验（prior）。

诸如正则化SGD之类的算法试图通过惩罚模型中变量的引入来去除一些可能的变量。在其他条件都相同的情况下，这些算法倾向于不包含变量。即便如此，要了解哪些变量可以剔除，这些算法也需要花费一定的时间。如果剖析模型时发现某些变量从未在模型中涉及，那么将这些变量剔除可能有利于加速后续的训练过程。类似地，不允许使用某些类型的变量之后会得到的新的模

型的版本，对这些版本进行测试非常有用。

通过剔除那些不携带信息的变量，Mahout中的其他分类器算法（如朴素贝叶斯和补充朴素贝叶斯）能够获益更大，这是因为这些算法本身无法通过正则化来剔除这些变量。

有一种情况相对比较普遍，就是剔除两个变量中的一个没有任何变化，但是如果同时剔除两个会导致性能显著下降。还有可能是单独使用某个特定变量跟做出随机决定的结果差不多。如果你觉得这个变量对性能影响微乎其微的话，那就把它剔除了吧。但是有时候即使该变量单独使用时看上去没有任何预测力，但是能够提高其他变量的预测性能。保守的说，删除任何一个没有出现在最终模型剖析中的变量可能都会带来正面影响，但是对于其他情况还是要具体问题具体分析。

2. 增加新的变量、交互和导出值

构建一个分类系统时，有些数据比其他数据更容易转换成可用格式。数据准备过程中的困难将会导致学习算法所需要的预测变量交付时间的延迟。相反地，有些变量可能会早于其他变量可用。在大型项目中，某些预测变量交付的延迟时间很容易就会上升到数周或者数月，而且有些预测变量很可能会被证明是无法获得的。

上述延迟的潜在可能性也意味着，无论有什么可用数据，建模和评估过程都要早点开始。一件常常发生的事情就是初始可用的预测变量已经足以构建一个可部署的模型。即使早期的模型在生产中可用，额外变量也可能会提高系统的系统，因此它们一旦可用，我们需要尽早对其进行测试。

另一个常见的新变量来源是对现存的连续变量的变形。将原始值的对数值或者平方值加入到系统中是值得的。广义的线性模型（如Mahout中基于SGD的模型）可以从这项技术中获益，如果不这样做的话，它们就只能关注预测变量和类别之间的线性关系。大部分基于树的模型不会注意到这种对连续变量变形带来的差异。

下面给出了对TrainingExample对象进行查询来获取行数的例子，之后行号采用三种不同的方式编码：

```
FeatureValueEncoder lineEncoder =
    new ConstantValueEncoder("lines");
FeatureValueEncoder logLineEncoder =
    new ConstantValueEncoder("log(lines)");          注意名称变化
FeatureValueEncoder lineSquaredEncoder =
    new ConstantValueEncoder("lines^2");

for (TrainingExample example : trainingData) {
  Vector v = new RandomAccessSparseVector(50000);

  double lines = example.getLines();
  lineEncoder.addToVector((byte[]) null, lines, v);
  logLineEncoder.addToVector((byte[]) null, Math.log(lines), v);
  lineSquaredEncoder.addToVector((byte[]) null, lines * lines, v);
}
```


通过对行数用多种方式转换变形，可以得到一个很容易具有多类推理能力的模型。例如，通过包含原始价格和出售价格的对数值，模型可以访问商品的折扣率。而如果不转换原始价格和出售价格，能够得到的只是绝对的折扣金额而不是原始价格的某个比值。到底哪种变形更好事先很难说，因此让学习算法自己选择可能是一个非常好的思路。

将变量的多个变形加入到朴素贝叶斯模型中没有意义，这是因为朴素贝叶斯只接受类别型或文本型输入。这些变量已经编码成二值数字，而对二值变量进行变形基本没有任何意义。

交互变量（interaction variable）是其他变量的组合，它们是分类器的另一力量源泉。例如，在一个市场应用中，将性别和品牌组合成交互变量可能效果会很好。如果没有这个交互变量的话，模型可能会推导出女人比男人对赠品更有或更没有反应，或者某个具体品牌更受或者更不受欢迎。然而，当使用性别和品牌组合在一起的交互变量时，模型可以推出某个特定品牌更受或更不受男人或女人欢迎。由于不同厂商会对其品牌进行专业化处理来吸引不同的细分市场，所以上述变量十分有效，这一点并不意外。通过散列特征表示，交互变量可以使用类似如下的代码来加入：

```
LuceneTextValueEncoder authorEnc = new LuceneTextValueEncoder("author");
LuceneTextValueEncoder subjectEnc = new LuceneTextValueEncoder("subject");
FeatureVectorEncoder authorPlusSubjectEnc =
      new InteractionValueEncoder("author_subject", author, subject);

for (TrainingExample example : trainingData) {
  Vector v = new RandomAccessSparseVector(50000);
  String author = example.getAuthor();
  String subject = example.getSubject();
  authorEnc.addToVector(author, 1, v);
  subjectEnc.addToVector(subject, 1, v);
  authorPlusSubjectEnc.addToVector(author, subject, 1, v);
  ...
}
```

这里，通过引用作者编码器authorEnc和主题文本编码器subjectEnc建立了一个交互变量编码器authorPlusSubjectEnc。包含作者名和主题行的字符串通过作者、主题以及它们的组合变量来编码。交互变量代表的不止是作者和主题，还包括它们相关联的事实，这区别于另一个作者使用本主题或者该作者讨论一个另外的主题。这允许分类器同时基于作者和主题来进行差异判别。如果不同用户使用同一文本来表示不同意义，那么如果不使用上述交互变量就无法学习到正确含义。然而利用交互变量，模型就可以将不同的意义归功于几乎相同的词语，这取决于它们的上下文。此外，由于同时包含了独立的作者和主题的编码器，模型也能不依赖于作者学到主题行的意义。

交互变量十分强大，但是它们会显著增加分类器的复杂度，使得特征编码和学习过程显著变慢。由于每个训练样本的特征数目都显著增加，这种做法也有可能导致精度的下降。

另一种对已知变量进行变形的做法是对一部分训练数据聚类，比如使用类似k-means算法对多个预测变量进行聚类。这种聚类会导致一个新的预测变量，即对每一部分数据而言，它的簇是一个类别型特征。

类似地，我们可以构建一个分类器模型，然后将该分类器的输出结果用作另一模型的预测变量。其思路是，第一个模型会学习问题的大致轮廓，而级联的第二个模型可以学习如何纠正第一个模型的错误。下面给出了上述做法的一个代码示例：

```
AbstractVectorClassifier subModel =
    ModelSerializer.readBinary(new FileReader("sub-model.model"),
    OnlineLogisticRegression.class);
ConstantValueEncoder subModelEnc = new ConstantValueEncoder("sub");
...
```

```
for (TrainingExample example : trainingData) {
  Vector v1 = new RandomAccessSparseVector(50000);
  ... encode features for subModel ...
  Vector v2 = new RandomAccessSparseVector(50000);
  double subScore = subModel.classifyScalar(v1);
  subModelEnc((byte[]) null, subScore, v2);
  ... encode other features ...
  ... train model ...
}
```

这里我们假设第一个模型已经训练好并放在文件submodel.model中。于是，在训练第二个模型时，我们构建一个特征向量作为第一个模型的输入，然后运行第一个模型得到一个得分，该得分被用作第二个模型的特征。

15.4.2 分类器调优

除了改变提供给分类器学习算法的数据之外，也可以对算法本身或者算法的定义参数进行修改。尝试这些方法的好处在于提供了多种方法通过有指导的反复试验来提高系统性能。

1. 尝试其他算法

尝试使用不同的分类器算法来看看到底哪个结果最好，这一点十分重要。Mahout支持朴素贝叶斯、补充朴素贝叶斯和SGD算法，除此之外，还有随机森林和支持向量机（SVM）的实验版本可用。

除了修改整个分类模型之外，还可以在保持模型的同时修改学习算法。具体地，基于SGD的分类器可以通过使用`CrossFoldLearner`交叉测试或者能够自动调节很多学习参数的`AdaptiveLogisticRegression`来运行低层学习算法。也可以从对数似然、AUC或者这些指标的混合指标中选择学习算法的优化目标。

我们可以基于性能、训练时间和分类速度对不同的算法进行比较。例如，可以使用朴素贝叶斯两个版本的算法和`AdaptiveLogisticRegression`。如果精度好的模型不止一个的话，就基于它们与系统其余部分的融合好坏来选择其中一个算法。

2. 调整学习算法

当具体使用SGD的低层学习算法时，有很多开关选项可以打开或关闭来获得不同的结果。当构建`AdaptiveLogisticRegression`、`CrossFoldLearner`或者`OnlineLogisticRegression`时，大部分参数的配置方法都很类似。这些方法概括在表15-5中。

表15-5 SGD学习类中的配置方法

算法参数	配置方法	在线Logistic回归（online logistic regression）	交叉折叠学习器（CrossFold Learner）	自适应Logistic回归（adaptive logistic regression）	解 释
`prior`	Constructor	Yes	Yes	Yes	设定正则化函数来鼓励稀疏性避免过拟合
`numFeatures`	Constructor	Yes	Yes	Yes	指定特征向量的大小
`numCategories`	Constructor	Yes	Yes	Yes	指定目标类别的数目

（续）

算法参数	配置方法	在线Logistic回归（online logistic regression）	交叉折叠学习器（CrossFold Learner）	自适应Logistic回归（adaptive logistic regression）	解　释
alpha	Setter	Yes	Yes		设定学习率计划
decayExponent	Setter	Yes	Yes		设定学习率计划
lambda	Setter	Yes	Yes		控制正则化
learningRate	Setter	Yes	Yes		设定学习率计划
stepOffset	Setter	Yes	Yes		设定学习率计划

在所有的SGD类中，先验（prior）、特征数目及目标类别数目都在构造函数中设置。AdaptiveLogisticRegression类通过在不同设置下并行运行并比较多个CrossFoldLearner的结果来控制学习率。这意味着AdaptiveLogisticRegression中唯一需要另外设置的参数用于控制学习率修改频率和并行CrossFoldLearners测试数目。对于CrossFoldLearner和OnlineLogisticRegression而言，需要指定学习率和学习率随时间衰减的方式。这可以通过使用编译器类型的方法来实现，比如：

```
lr = new CrossFoldLearner(targetCategories, numFeatures, new L1())
          .lambda(currentLambda)
          .learningRate(initialLearningRate)
          .decayExponent(0.5)
          .stepOffset(2000);
```

通常，decayExponent和stepOffset用于控制初始学习率随时间下降的方式，而设置alpha值则要少见得多。

除了上述设置之外，CrossFoldLearner还允许设置其内部所使用的交叉检验次数。在差不多所有情况下默认值5都是可以接受的。使用比如2或3这种更小的值，会将训练中的数据数目降低到大部分应用可以接受的次数之下。而将该值设置得比5大会导致更大的计算开销，因为此时对每次交叉都需要一个独立的学习器。

一个最简单的调整所有上述参数的办法是仅仅在默认设置下使用AdaptiveLogisticRegression。但是这需要花费相当多的计算开销，这是因为AdaptiveLogisticRegression内部使用了20个CrossFoldLearners构成的进化池，每个CrossFoldLearners维护5个OnlineLogisticRegression对象。这意味着必须花费一定开销在同一时间运行100个学习算法。

尽管如此，在实际当中上述做法却并没有那么严重，这是因为从磁盘读取训练样本并将它们转换成特征向量就与运行100个学习算法的代价相当。另外，在实际情况下，一个学习算法将至少在数十个或者成百上千种参数配置下运行。而AdaptiveLogisticRegression却能自动实现这一点。这样做的一个副作用是，数据必须只能做一次转换。不好的选项早期就会被去掉，因此不会有计算开销浪费在它们身上。

15.5 小结

本章当中我们学习了Mahout中计算分类质量指标的主要API以及如何使用这些指标的基本知识。我们还学习了常见的漏洞如何影响这些指标以及如何发现并对常见问题进行纠正。

正如本章例子指出的那样，分类器的问题可能十分隐蔽。不像大部分计算机程序一样，即使严重的错误也有可能不会导致完全的崩溃，这是因为学习算法常常学习如何从部分崩溃中进行恢复。这种情况既是好消息也是坏消息，这使得细致的分类质量检测十分重要。

评估并不只是构建和调整高性能分类器的一个迭代过程，它也是在产品上线之后的一个持续的重要步骤。不断的性能评估对于性能的维持相当重要。外部条件的变化可能导致模型的可用性降低，或者数据集上的处理错误会降低模型的有效性。任何情况下，生产中的持续评估对于意识到是否需要重新调整或者是否重新训练模型或者选择新方法都十分重要。而这些方面会在下一章的分类器部署中进行探讨。

第16章 分类器部署

本章内容

- 指定分类器的速度和规模需求
- 构建大规模分类器
- 交付一个高速分类器
- 构建并部署分类服务器

本章主要考察在将分类系统变成实际产品时所面对的问题。前面的章节中，我们讨论了很多问题，包括面对大规模数据集时使用Mahout分类器的优点、这些分类器如何训练、它们内部是如何工作的以及如何对这些分类器进行评估等,但是这些讨论都有意省略了将分类器做成产品的许多实际问题。本章将在解释如何部署一个高速分类器时考察这些实际问题。我们会介绍优化特征提取和向量表示的技巧，会描述如何在负载和速度之间折中，还会解释如何构建超大规模系统的训练流水线。最后，我们会提供一个基于Thrift的服务器的部署案例，该服务器允许全功能负载均衡分类。

如果打算在产品中使用Mahout，那么很有可能下面某些或全部条件成立：拥有极大规模训练数据，要有高吞吐率，需要快速响应，或者需要和现有大型系统集成。上述条件也意味着在建立系统时需要具体考虑的架构和设计因素。大数据集需要高效的读取和组织，高吞吐率需要可扩展且优化的代码。集成到大型系统需要一个简单清晰的作为网络服务的API。Mahout提供了强有力的办法来处理大数据系统，但是也有些办法来简化或重述问题以使像Mahout一样的工具更加强大。

下一节中，我们将概述如何设计、构建和部署一个实际的大规模分类器。

16.1 巨型分类系统的部署过程

当部署基于Mahout的分类器系统时，有一些标准步骤可以用于保证部署的成功性。其关键在于事先预测可能出现的问题并从中选择那些需要集中精力避免的问题。对于这些问题的具体处理策略将在16.2至16.4节中讨论，本节主要介绍部署巨型系统的过程。

注意 为理解Mahout对你的分类项目带来的潜在价值，要了解你的系统的各个方面，包括系统是否特别大，或者是否要求系统运行特别快。这些关键点将确定Mahout是否正确的解决方案。

整个部署流程可以分成如下几步：

- 理解问题；
- 根据需要优化特征提取过程；
- 根据需要优化向量表示；
- 部署可扩展的分类器服务。

上面的每一步都在本节中描述然后应用到随后的案例中。

16.1.1 理解问题

第一步要确定问题的真正规模以及在训练和生产环境中所需要的分类速度。训练数据的规模会决定特征提取的构建过程，比如到底是采用诸如关系数据库的传统工具来构建这个模块，还是使用类似Apache Hadoop的工具来构建一个高扩展性的特征提取模块。有关规模和速度的需求细节将在16.2节讨论。

训练所允许的时间将确定是否能够使用一个像SGD一样的串行训练分类器。具体原因如下：如果使用并行算法而训练时间的要求又有限，那么当处理更大规模数据系统速度不够快的情况下，通过增加机器就能满足需求。但是，如果使用串行算法，只有对单台计算机固定时间段的安排，因此只能通过额外处理加速解析数据和编码过程来达到要求。

生产环境下分类器的吞吐率和延迟性需求将确定是否需要一个分布式的分类器，以及是否需要进行大量的预计算来最大化分类器的速度。对于一定延迟开销下的极端吞吐率需求，需要确定的事情还包括是否需要进行批量分类。

16.1.2 根据需要优化特征提取过程

当样本规模高于数千万时，采用诸如MapReduce的这种可扩展技术来进行特征提取就变得越发重要。

如果规模很小不到几百万，那么任意技术都可行，那些诸如关系数据库之类的久经考验的数据管理和提取技术都能处理得非常好。

当规模更大时，就可能必须要采用类似Hadoop的工具来构建并行的提取架构，或者交付给像Aster Data、Vertica或Greenplum之类的商业支持系统。如果使用Hadoop的话，那么还需要确定特征提取代码的具体实现方案。这些可选的方案包括Apache Hive（http://hive.apache.org）、Apache Pig（http://pig.apache.org）、Casading（http://www.cascading.org）甚至Java写的MapReduce原始程序。

这里的一个关键点是任意下采样过程都要作为特征提取而不是向量表示的一部分以保证取样过程可以并行处理。实际针对大数据系统的特征提取相关问题的细节将在16.3节介绍。

16.1.3 根据需要优化向量编码

如果对于每个分类器在经特征提取和下采样之后不到100万样本，就可以考虑一个诸如SGD的串行模型。如果这个数字超过500万，那么就要考虑对数据解析和特征向量化代码仔细优化。

这种需求可能涉及不带字符串转换的分析过程、多线程的输入转换以及缓存策略等，这些方法可以以最快的速度，给学习算法提供数据。

如果训练样本数超过1个亿，可以考虑直接使用带预定学习率参数的CrossFoldLearner类。如果问题相对稳定的话，可能可以采用某种自适应策略来学习学习率然后选择其中较好的结果。使用AdaptiveLogisticRegression来实现上述策略的好处在于可以在学习中少用20倍的数值运算，不足之处在于加速比并没有听起来那么大，并且整条学习链需要更多的维护工作。

16.1.4 部署可扩展的分类器服务

在大部分应用中，将分类器分隔成一个独立的服务来处理数据分类请求比较方便。这些服务是典型的基于网络RPC协议的常规服务，这些协议包括Apache Thrift或基于REST的HTTP等。16.4节给出了一个基于Thift构建的分类服务的例子。这种做法特别适合于需要对每个输入样例执行许多分类操作的情况，比如当对许多对象进行测试以得到期望值的情况就是如此。

在有些情况下，网络往返的延迟开销无法接受，此时分类器就必须作为一个直接调用库来集成。当非常细粒度的需求需要在极端速率下处理时，上述情况就会发生。由于现代网络服务架构能够支持每秒钟超过50 000的吞吐率，因此上述情况相对比较罕见。

16.2 确定规模和速度需求

构建分类器第一步就是要确定规模因素，包括训练数据的数量和系统要达到的吞吐率。从多个维度了解问题的规模可以指导设计中的一些决策，这些决策会刻画即将构建的系统的各个方面。

16.2.1 多大才算大

如果考虑使用Mahout来构建分类系统，首先问一下自己有关系统的几个如下问题，然后根据问题的答案来确定系统的哪些部分需要采用大数据技术，而哪些部分只需要采用传统技术处理。

- 有多少训练样本？
- 分类的批处理规模多大？
- 分类批处理要求的响应时间多少？
- 每秒钟必须要处理的分类次数总共有多少？
- 系统期望的最大负载是多少？

最大负载的估计

为估计最大负载，可以简单地假设每天有20 000秒而不是真正的86 400秒，这样处理起来比较方便。这里使用一个缩水的仅由20 000秒构成的一天，可以在合理的事务突发性水平上从平均值转换成峰值，从而完成峰值的估算。

对长期的平均事务率进行估计通常十分容易，但是要设计系统时必须知道其所支持的峰值。利用上述概算法则，可以使用一个称为费米估计（Fermi estimate）的方法来估计系统每秒钟必须要处理的事务数目。恩里克·费米（Enrico Fermi）是著名的物理学家，特别以擅长类似上述的估计而闻名于世，于是他的名字和这些估算关联在一起。

上述值并不需要精确地估计。通常而言，只要估计的值与真实值在同一数量级内就足以用于初始系统的规模和架构设计。当转向产品时，将会最终重新调整上述估计值。

一旦你知道会使用的训练样本数目，就可以考虑Mahout是否适合你的项目，并在适合的情况下选择好的算法。

1. 训练样本数的含义

通常来说，当训练样本一旦超过100 000时， Mahout就会变得有意思起来。但是能够处理的数据规模并没有下界。当有上百万或更多训练样本时，大部分其他的数据挖掘方案无法完成训练过程，而当训练数据规模达到上亿或者上十亿时，就很少有系统能够构建可用的模型。训练样本的数目以及模型的类型是训练时间的主要决定因素。

对于一个像Mahout一样的可扩展系统来说，可以通过小规模数据上的多次测试对期望的训练时间进行可靠推断。对于像SGD一样的串行学习算法来说，学习时间应该与输入规模呈线性关系。实际上，对于很多模型来说，会在远远没有处理完所有输入之前就会收敛到一个可靠的错误水平上，因此训练时间和输入规模呈亚线性关系。

如果使用并行学习算法，比如朴素贝叶斯的并行版本，那么学习的时间大致与输入规模除以学习用MapReduce节点数之后的结果呈线性关系。由于朴素贝叶斯的学习过程编写成一个批处理系统，即所有输入会经多步才能处理，所以该过程不能很早中止。这也意味着即使部分数据虽然足以生成一个可用的模型，而这里必须要处理所有的训练数据才能得到一个可用的模型。

2. 分类批处理的规模

一个常见的情况下，对于单个请求需要执行一系列分类过程。这种需求常常发生于多个不同对象需要评估的情况下。比如，在一个广告投放系统中，通常有数千个广告，每个广告都有模型需要评估来确定哪个广告出现在某个网页中。这里的批处理规模就是需要评估的广告数目。有些应用的批处理规模十分大，超过100 000个模型。

另一个极端情况是，某个欺诈检测服务器可能只有一个模型要进行值计算，由于单个事务是否存在欺诈只涉及一个决定，即存在欺诈与否。这种情况和上述广告选择服务器存在很大的差异，后者当中对于多个可能的广告的每一个都需要对点击模型进行计算。

3. 最长响应时间和所需吞吐率

最长响应时间确定单个分类批处理完成所需要的时间。该时间也确定了所需分类速率的下界。当然，这个值只是一个下界，这是因为如果每秒钟处理的事务数目足够的话，就有可能支持更高的速率。

通常来说，在较小的分类批处理规模下很容易就可以构建每秒1000次分类的系统，这是因为这个处理速度相对于当前计算机的处理能力而言要小一些。实际上，本章后面会介绍的一个基于

Thrift的服务器很容易就能达到这个需求。直到批处理的规模超过100到1000时，该服务器都能够基本支持上述事务率，这是因为此时大部分的开销都是网络开销。如果批处理规模达到10 000或更多，响应时间将开始攀升。如果批处理规模更大并且要求亚秒级响应时间， 那么可能要对分类系统横向扩展或者重构。

16.2.2　在规模和速度之间折中

使用Mahout的一个好处是，相对于非可扩展系统来说，Mahout可以更宽松地支持在数据规模和处理速度之间折中。也就是说，在系统的多个方面之间通常需要某种程度的折中， 因此需要考虑如何对它们进行折中。为达到此目的，需要牢记的是，你的系统的某个方面的需求可能与其他系统不太一样。在同一系统中使用不同的分类器来完成不同目标是一件很常见的事情。

一旦系统的各个方面都考虑得比较清楚，就可以开始规划部署，并考虑哪些方面需要特别注意，哪些方面需要快速实现。下面的章节给出了部署过程的每一步所需要重点考虑的事项。一般来说，这些注意事项要不在分类训练过程中应用，要不在分类部署之后应用。

1. 训练

为建立可分类数据所需要运行的大规模联结运算常常占据了训练的大部分时间。用户可以使用标准的Hadoop工具来实现这些运算，但是必须注意要使得联结运算高效运行。

对负例样本进行下采样（downsampling）会大大降低训练样本的规模。在欺诈检测和点击优化中，非兴趣训练样本数目往往大大超过兴趣样本（欺诈或点击）数目。保留所有的兴趣样本对于精确学习至关重要，但是只要仍然保持非兴趣样本比兴趣样本足够多，就可以随机丢掉一些非兴趣样本。这种样本的丢弃过程就称为下采样，该方法常常作用联结过程的一部分进行处理。

在获得训练样本之后，特征的表示过程常常占据了训练的大部分时间。训练数据的下采样也降低了特征表示的时间，但为使表示过程更加高效仍然有很多事情要做。

至始至终我们都要牢记这一点，即建立一个好的分类模型需要多遍的反复调优过程，必须确保训练流水线能够使得上述迭代过程高效进行。16.3节介绍了训练流水线中的架构和策略性问题。

2. 分类

当准备将分类器集成到系统时，总体的分类吞吐率需求将确定如何对分类器进行优化。大部分Mahout分类器都足够快，因此只有在极端系统中才需要考虑上述问题。但是需要检查此时是否处于极端系统当中。如果不在，那么就可以直接采用常规方案处理，而如果在极端系统当中，就必须要安排时间来拓展通常的性能限制。

如果每次事务当中只需要做少量的分类处理，并且只有一般的延迟时间和吞吐率需求，利用每个事务单线程的传统服务设计就足以达到要求。基于REST的服务或许可以接受，或者采用性能稍微高一点点的网络服务技术，比如Thrift、Avro或者协议缓冲区（Protocol Buffer）技术。16.5节给出了如何构建这样的一个服务的例子。

如果每次事务当中所需要的分类处理次数达到中等级别（数十到数千），并且需要更高的延迟性和吞吐率要求，那么需要采用一个基于多模型或多输入的多线程过程来对基本的分类服务进行增强处理，但是此时在其他方面基本的设计仍然足以满足需求。

当碰到十分极端的批处理规模需求需要在每个服务事务中处理大量的模型计算时，就必须深入到模型评估的核心代码中来将模型评估分成多个部分，有些部分可以提前评估，而有些部分必须要在分类时评估。这种优化过程非常针对于特定的应用系统，其内容远远超过本书的覆盖范围。这种情况下使用Mahout的一个实现案例在第17章给出。

16.3 对大型系统构建训练流水线

分类系统需要训练数据，而可扩展的分类系统可以处理非常大规模的数据。为了给这些系统提供数据，必须要能处理更大规模的数据，这样才能从中选出一些来训练分类器。

为了处理这么大规模的数据，需要牢记下面几件事。首先也是最重要的事情是保持技术和任务之间的匹配。比如，如果训练样本不到100万，那么几乎所有的技术都能奏效，包括传统的关系数据库甚至对数据进行扁平文件表示的脚本。而当训练样本达到1亿，关系数据库仍然可以进行必要的数据准备工作，但是获取数据会逐渐变成一件越来越令人痛苦不堪的事情。如果样本规模达到10亿或更多，关系数据库不再实用，可能必须要转向诸如Apache Hadoop之类的MapReduce系统。

10亿训练样本听起来很极端，但是在实际中达到这个水平的案例却出奇普遍。例如，在17章即将提到的Shop It To Me公司一天会发送几百万邮件，而每封邮件包含数百商品。因此，它每天能够产生数亿的训练样本，并且很快就能达到数十亿规模。所有大型的广告网络每天的广告展示数目远高于10亿，每次展示就是一个潜在的训练样本。一个每月拥有百万独立访问者的音乐流媒体服务大概每月能产生10亿的训练样本。

不管所处理数据规模多大，模型训练流水线必须要支持如下功能：

- 必须保持数据记录获取和存储的一致性，通常它们基于时间来分割；
- 数据记录必须要和某种参照数据进行联结运算来增强；
- 数据记录必须要加入到目标变量值中；
- 训练数据的下采样可能有用应该支持；
- 除了简单的下采样之外，其他一些过滤和投影操作也应该支持；
- 训练数据必须要采用训练特征来表示；
- 训练后的模型必须要保留关联的元数据信息，这些信息描述了对数据进行表示的方式。

训练数据流水线的多个部分往往可以横向扩展，但是当涉及一个模型的训练时，一些极易部署的Mahout模型并不适合在训练过程中横向扩展。这也意味着，当数据规模很大的情况下，早期的数据清洗和处理可以采用高度可扩展的方式来处理，但是当模型训练完毕之后，就必须要对数据的高效表示下大功夫。

这一节将介绍利用Mahout及相关技术来构建满足上述功能的数据流水线时，所要考虑的多个重要问题。接下来的介绍中假设读者已经熟悉在小规模情况下所用的技术，因此只关注那些在大规模条件下的技术。

16.3.1　获取并保留大规模数据

训练大规模模型意味着将不得不处理真正的大训练数据集合。像以前一样，那些在小规模数据微不足道的问题往往在大规模上数据上一下子变成很严重。大部分的这些问题都源自一个事实，即大规模数据拥有大量在比喻上被称为惯性（inertia）的东西。这种惯性使得大规模数据的移动或转换十分困难，需要为将来的打算做好事先的规划。如果对于新数据没有直接的体验，上述规划不可能高效完成。因此，在对数据有足够体验之前，用户需要回到并遵循最佳实践标准。

1. 获取数据

大规模数据集上的第一个问题就是对它们的获取。这听起来很容易，但是就大规模数据源上所需要做的事情而言，它比看上去要更难。这个处理层次的主要做法是保持数据的相对紧凑性，并保持数据产生过程中的很自然的内在并行机制。

例如，如果拥有大量服务器，每台服务器每秒处理数千事务，那么此时采用单个中心日志服务器可能是个糟糕的做法。这是因为极度的中心化易受单点故障的影响。数据获取系统很容易反向影响整个所观察系统的运行。为避免上述问题，在数据获取过程中进行减载至关重要，宁愿丢失一些数据也不能让生产系统崩溃。

第二个获得大规模数据集的技巧是，确保保存数据的目标是为了让机器以机器可读的格式来读取这些数据。拥有对人可读的日志也很好，但是从这些日志中提取数据是常见的错误源之一，这是因为这些数据的格式过于宽松。最好将人要读的数据保存为对人可读的格式而将要处理的数据保存为机器可读的格式。

对于大规模数据获取来说，可以使用多种不同的数据格式，但是只有其中的一部分可以同时提供紧凑性、快速解析库和最重要的schema的灵活性，而这些要求是必要的。Schema灵活性可以允许所存储数据的演变同时仍然能够读取旧数据。

两种最突出的同时支持上述功能的编码格式是Apache的Avro和谷歌的Protocol Buffers。两种格式的支持都非常好，并且提供可比的功能和灵活性。利用逗号或Tab分开字段的数据存储方法非常普遍，但是这种方法当schema演变时很快就会出现问题，这是因为哪列数据包含哪个字段会出现混淆。当不同schema的文件必须要同时进行处理时，上述混淆会变得特别严重。

2. 分割并存储数据

当获取并存储数据时，必须要遵循如下组织原则来保证最终得到的是可扩展系统。

- 通常情况下，每个目录下包含的文件数目通常应该不多于几千。更大的目录会导致扩展问题，这是因为在极大的目录下打开和创建文件将需要更长的时间。
- 程序处理的文件数目应该尽量少，这可以通过不处理无关数据并且将文件保留得尽量和实际一样大来实现。避免无关数据的处理会限制数据的传输量，而使用大文件会最小化磁盘寻道带来的时间损失从而最大化I/O速度。
- 文件组织的目录结构应该支持常用的相关训练数据选择方式。目录组织得好可以在不遍历所有文件的情况下发现相关文档。
- 文件通常应该不超过1GB以方便处理。文件应该足够小以方便在合理的短时间内在系统之

间传输。这也使得可以通过反复快速传输来限制网络中断带来的影响。1GB左右是个很不错的中间妥协值。

遵循上述原则通常会导致如下结果：顶层目录存储的是数据的一般类别，而子目录下包含逐渐精细的按照时间划分的结果。

提示 将多个小的旧文件合并成能够代表更大时间段的大文件通常是一个好的做法，这可以避免后续处理过程有过多的文件输入。如果对于最近的时间段需要更细的时间粒度，那么文件可能会按照非常短的时间段来保存，比如5分钟间隔，但是几天以后，这些文件就应该按小时然后按天累加到文件中。如果每天的数据超过1~2 GB，那么每天保存多个文件或许是一个好的做法。

---

3. 增量式处理

在很大程度上说，特征提取并不是那种需要在长时间段上进行累计的任务。这也意味着一旦对一小时的数据进行了必要的联结运算和数据缩减处理来得到相应的训练数据，该数据在下一个小时的数据到来之前可能不会改变。这种特性使得每次训练完成时增量式获取训练数据相当容易，而不是为全时间段进行所有的特征提取。

同样，这种增量式处理过程也使得短时间段内所提取的特征可以累积，然后合并后长期存储。例如，将以小时计的训练数据进行累积并将它们合并成以天计的训练文件然后几乎可以实现永久保存，这种做法十分常见。这也很好地对应了原始数据的常见保存方法。

当分类系统相对较新时，新特征的加入和下采样中的变化可能意味着需要时常重构训练数据文件。随着时间的漂移，变化的频率通常会下降。一旦这种情况发生，增量式构建训练数据就会是一个赢点，这是因为数据到达和模型训练开始之间的延迟会被降低。

16.3.2 非规范化及下采样

在用于训练模型之前对直接获取的数据进行非规范化处理相当普遍，这是因为原始数据可能来自于不同的存储格式，包括日志文件、数据库表和其他数据源的格式。在非规范化处理中，数据可以低冗余地存在独立的数据表中，这些表通过键互相联结以使得每条记录都是自包含的而不需要指向外部数据，这些表的例子包括用户信息表或产品描述表。

由于模型训练算法并不能对外部参照数据进行解析，所以它们的输入必须要完全非规范化才可用。非规范化几乎往往通过某种联结运算来完成。在小规模到中等规模的数据集上，几乎所有的方法都可以足以完成这些联结运算，像关系数据库一样的工具相当有用。然而对于大规模数据集，这些联结运算很难高效处理。此外，在进行联结运算时必须要考虑数据的特点。

1. 首先联结目标变量

几乎到处都需要联结运算将目标变量值和预测变量关联起来。如果可能的话，目标变量应该在非规范化之前关联，这是因为常常需要根据目标变量的值来决定训练样本的下采样做法。在任一后续联结操作之前进行下采样，由于在更少的数据上得到几乎一样的结果，所以通常能实现训

练中大量开销的节省。

2. 内存中联结

某些情况下，训练数据需要联结成行数有限的二级表。如果情况如此，二级表有时可以调入内存，通过查询二级表的内存表示来完成联结。

内存表的装载过程在`Mapper`或`Reducer`的`configure`方法中实现。内存中的联结运算可以在一个MapReduce的程序内部作为`Mapper`或者`Reducer`的一部分来完成，尽管正在生成分类训练数据，Map端的联结仍然更普遍。

在某些参考文献中，内存式联结也称*内存支持式联结*（memory-backed join），比如Lin和Chris Dyer的书*Data-Intensive Text Processing with MapReduce*中就是这样。

3. 合并联结

当待联结的两个数据集都很大的时候，可能需要合并联结（merge join）操作。在一个顺序执行非并行程序实现的合并联结操作中，两个基于相同键排序的数据集可以在两个数据集的单遍扫描中完成联结。

在一个MapReduce程序中，由于不太可能按照每个Map正确合并数据的方式划分两个数据集，因此情况有点复杂。然而，对一个文件进行排序时可以建立索引。索引数据的路径可以作为边界数据提供，从而每个Map个过程一开始读取非索引数据的一个划分子集，然后在索引数据中的正确位置附近寻找并从该位置开始合并。

当数据本来就排好序时，合并联结十分有用，但是常常被忽略的一点是，如果输入数据集中只有一个排好序并被索引的话，那么合并联结可以在`Reducer`而不是在`Mapper`中完成。这种做法可以允许利用MapReduce框架来对非索引输入进行排序。而相对于全Reduce联结来说上述做法是否节省时间只能通过实验来确定。

4. 全Reduce联结

从编程量来说，最简单的联结数据的方法是进行全Reduce联结。这也意味着在`Mapper`中只需要简单地读入两个数据集然后按照感兴趣的键进行归约即可。这种做法可能会比合并联结的花销更大，这是因为在Hadoop的混洗（shuffle）步骤中需要对更多的数据进行排序。

当进行全Reduce联结操作时，Hadoop对结果进行排序将很有帮助，这样训练记录会在加入到训练记录并非规范化之后到达。利用Hadoop进行排序可以允许`Reducer`采用全流的方式来编写。由于Hadoop中的混洗和排序进行了高度优化，一个暴力的全Reduce联结可能起码和合并联结一样高效，特别在进行一个Reduce端的合并联结时更是如此。

如果训练数据规模大于要联结到的数据，那么全Reduce联结和任意一种合并联结可能在速度上都相当。

16.3.3　训练中的陷阱

即使假设数据如你所想、联结工作正常、数据解析也正确，仍然存在一些非常容易陷入的陷阱。其中最隐蔽、诊断也最困难的是目标泄漏。另一个常见的问题是所使用的数据编码表示不合理，从而导致在期望的结果和模型看到的数据之间存在语义失配。下面将详细讨论这些问题。

1. 目标泄漏或目标泄漏

保持每天的训练文件能够充分促进随时间变换的训练过程，这种训练过程通常对于避免目标泄漏十分必要。

例如，假定在某个目标定位系统中想用的特征是基于点击历史的用户聚类结果信息。如果使用该聚类结果来训练出同一时间段的一个模型，或许会发现最终模型在对点击进行预测时效果非常好。然而，这里发生的事情是，基于点击历史的聚类可能会将没有点击的人与有点击的人放到不同簇中，这种划分方法意味着，在聚类结果建立时用户簇就是一个目标泄漏。

图16-1给出了上述目标泄漏潜藏在设计当中的一个示意图。为修复这个漏洞，不能仅仅只是删除出问题的变量。基于用户历史的聚类仍然是一个有价值的预测变量。这里的问题只是因为模型没有在新数据上进行训练。

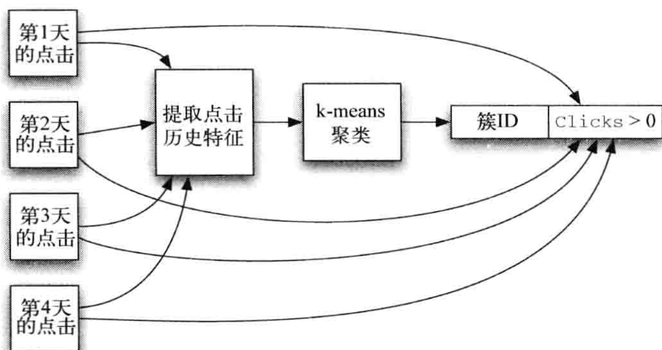


图16-1 注意不要这么做：由于目标变量（Clicks > 0）基于簇ID所在的相同数据，因此基于点击历史的聚类会在训练数据中引入一个目标泄漏

图16-2给出了如何先通过对早期数据聚类然后利用近期数据训练来克服上述问题的做法，其中近期的数据对聚类算法来说是未知的。更进一步，留存测试数据可以来自更近的数据。

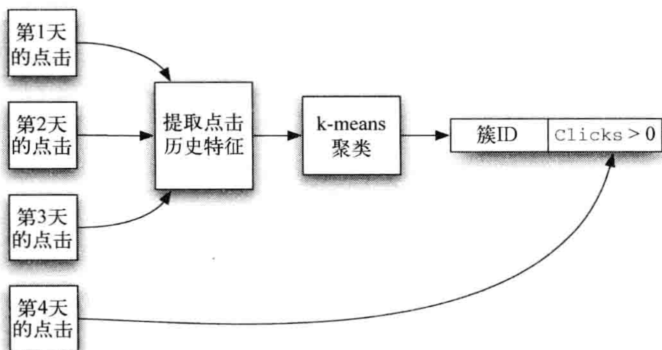


图16-2 一种避免目标泄漏的好方法：基于前3天的点击历史计算簇，然后从第4天开始推导目标变量（Clicks > 0）

按时间分开保存数据能够使得获取测试数据非常容易。在这种场景下，可以利用第五天的数据作为留存数据来评估模型的性能。

2. 数据表示中的语义失配

玫瑰即使不叫玫瑰，依然芳香如故。但是对于训练数据来说，整数并不总是表示看上去那样的意义。某些情况下，整数代表的是两件事之间间隔的小时或天数。另外一些情况下，整数代表类别的编号。上述情况的区别已经在前面章节中讨论。然而，尽管我们愿望良好，训练模型中整数数据的非正确处理仍然十分普遍。由于非正确数据编码表示的模型实际上有可能在在很大部分时间内性能表现正常，所以这种错误可能很难调试。

这种错误的诊断关键是，对最终模型的内部结构进行考察，来寻找应用到每个字段的多个权重。如果发现某个字段的具体值上有大量权重，那么该字段被看做是类别型或单词型值的标识符（ID）。如果发现某个字段只有单个权重，那么该字段被看做是连续型变量。不管模型如何看待字段，其方式应该和我们看待字段的方式一样。

上述规则的一个例外发生在拥有的整数确实是整数但是不同整数值的个数有限时。将这类变量看成是类别值可能会非常有用，即使它们看成连续值更正确。只有通过实验才能真正确定上述做法是否有效，但是只有整数的取值数目很小时，才值得尝试一下上述的小技巧。

16.3.4　快速读取数据并对其进行编码

对许多应用来说，前面章节中介绍的数据读取、解析并表示成向量的方法已经够用。然而，需要注意的是，Mahout中的SGD分类器只支持单机而不是多机并行下的训练。这个限制就会导致大规模训练数据下的训练时间很长。看一下剖析器（profiler）就会发现Mahout SGD模型训练API中的大部分时间都花费在训练数据的准备而不是真正的学习过程中。

为加速起见，必须要深入到比平常Java程序更低的低层。例如，`String`数据类型非常方便，它内置了Unicode的支持，并且拥有很多用于解析、正则表达式匹配等操作的库。但不幸的是，上述令人愉快的通用性同时也付出了高复杂性的代价。对于训练数据而言，字符集和内容上的假设限制通常要比一般文本严格得多。

训练程序遇到的另一个主要的速度瓶颈来自于处理中的复制模式。在这种模式中，常常通过构造不可变字符串（immutable string）代表输入行来处理数据。这些行又划分成多个分别代表输入字段的新的不可变字符串。于是，如果这些字段表示的是文本，它们又会在词条化过程中划分成多个新的不可变字符串。而词条可能进行了词干还原处理，而这一过程又要涉及另一些不可变字符串的生成。

在上述重复制的编程模式下，完成所有的字符串内存分配需要消耗很大的代价，很多人都集中关注通过广泛重用数据结构来减少分配开销。然而，实际开销中的内存分配开销并没有那么大，主要开销是反复复制的开销。 另一个开销源自Java在构建新的`String`对象时涉及的对每个新字符串的散列编码构造过程。散列编码计算的代价几乎和数据复制的代价一样大。

上述复制也为大规模加速提供了机会，我们可以通过更低层抽象和避免所有复制来实现加速。在更低层处理的模式下，如果要分配新结构时，可以试图来引用而不是复制原始数据。这些

小结构的生命周期同时也十分短暂，因此它们的分配和收集消耗几乎为零。

下面给出了一个字符串复制模式的例子，其中给出的是一个简单的数据分析类，它利用字符串来解析由制表（TAB）或逗号分隔的数据。该例子展示了一个简单、清晰但较慢的解析此类数据的方法：

```
class Line {
  private static final Splitter onTabs = Splitter.on(SEPARATOR);
  private List<String> data;
  private Line(String line) {
    data = Lists.newArrayList(onTabs.split(line));
  }
  public double getDouble(int field) {
    return Double.parseDouble(data.get(field));
  }

}
```

很显然，上面的代码没有给出重要细节，但是思路非常清晰。上述类将代表一行的字符串划分成一个字符串列表。这种划分和新字符串的分配过程需要对数据反复复制。此外，字符串复制中，每个字符需要复制两个字节。

下面的代码清单中给出了如何利用上述代码来从包含CSV数据的文件中解析和表示数据的过程。

代码清单16-1 解析和表示CSV数据的代码

```
public static void main(String[] args) throws IOException {
  FeatureVectorEncoder[] encoder = new FeatureVectorEncoder[FIELDS];
  for (int i = 0; i < FIELDS; i++) {
    encoder[i] = new ConstantValueEncoder("v" + i);     <-- 每个字段构建一个编码器
  }
  long t0 = System.currentTimeMillis();
  Vector v = new DenseVector(1000);
  BufferedReader in = new BufferedReader(new FileReader(args[1]));
  String line = in.readLine();
  while (line != null) {                                  重置向量内容
    v.assign(0);                                      <--
    Line x = new Line(line);                <-- 解析字符串
    for (int i = 0; i < FIELDS; i++) {
      encoder[i].addToVector((byte[]) null,            <-- 对每个字段进行编码
          x.getDouble(i), v);
    }
    line = in.readLine();
  }
  System.out.printf("\nElapsed time = %.3f s\n",
        (System.currentTimeMillis() - t0) / 1000.0);
}
```

上述代码以每次一行的方式读入数据，然后利用前面的Line类对每行进行解析。对发现的每一个字段，用一个特征编码器将该字段加入到特征向量中。每个字段都有一个独立的编码器。当要解析百万行数据而每行包含100个数据元素时，上述代码清单中通过字符串解析和编码表示的做法需要在一台2.8GHZ Core Duo（英特尔双核处理器）处理器的机器上运行75~80秒钟。如果

做一些改进，速度至少可以提高到原有的5倍。通过下面将要介绍的改进方法，完成相同的任务的一个程序只需要不到15秒。

1. 分割字节，而非字符

加速的第一步是使用大规模字节的I/O，并且不将字节转换成字符串从而避免对输入片段的反复复制。在`Line`类中，每一行从输入中复制，然后当转换成字符串时又一次进行复制，再接着在解析出每个片段时再次复制。上述所有过程使得程序容易理解和调试，但是肯定降低了运行的速度。

对很多数据文件来说，避免到Java字符串的转换是一件好事情，这是因为文件通常都有一个可以通过ASCII字符集来表示的限定数据格式，其中每个字符通过单个字节来编码。而Java字符需要两倍的字节空间，移动它们需要两倍的内存带宽。更进一步，通过字节数组，可以几乎完全避免复制操作。如果正在考虑的字段包含数字，那么使用常规的Java原语意味着这些字段将被一些例程进行解析，这些例程能够将通常的Unicode字符串转换成数字。由于基本的数字在Unicode中出现多次，因此上述转换并不像听起来那么容易。如果所有数据都是ASCII格式，那么上述转换要比先将数据转换成Unicode然后再将这种通用表示结果转换成数字要简单得多。

代码清单16-2给出了一个`Line`类的修改类`FastLine`，它使用`ByteBuffer`来避免复制。该代码也是Mahout样例中`SimpleCsvExamples`程序的一个内部类。`Fastline`也采用一种高度专用的解析方法来解析数字，它能解析由1、2位ISO拉丁字符集数字构成的整数。

代码清单16-2　字节级CSV解析代码

```
private static class FastLine {
  private ByteBuffer base;
  private IntArrayList start = new IntArrayList();
  private IntArrayList length = new IntArrayList();
  private FastLine(ByteBuffer base) {
    this.base = base;
 }
 public static FastLine read(ByteBuffer buf) {
    if (buf.remaining() == 0) return null;
    FastLine r = new FastLine(buf);
    r.start.add(buf.position());
    int offset = buf.position();
    while (offset < buf.limit()) {
      int ch = buf.get();
      switch (ch) {
       case '\n':
         r.length.add(offset - r.start.get(r.length.size()) - 1);
         return r;
        case SEPARATOR_CHAR:
          r.length.add(offset - r.start.get(r.length.size()) - 1);
          r.start.add(offset);
          break;
        default:
      }
    }
    throw new IllegalArgumentException(
          "Not enough bytes in buffer");
   }
```

避免装箱（boxing）的特定集合（collection）

记得这里是引用

记得行尾即字段结尾

注意下一个字段开始

假定缓冲区中包含完整的一行

```
    public double getDouble(int field) {
      int offset = start.get(field);
      int size = length.get(field);
      switch (size) {
        case 1:
          return base.get(offset) - '0';
        case 2:
          return (base.get(offset)-'0') * 10+base.get(offset + 1) - '0';
        default:
          double r = 0;
          for (int i = 0; i < size; i++) {
            r = 10 * r + base.get(offset + i) - '0';
          }
          return r;
      }
    }
}
```

上述代码清单中Fastline类使用 Mahout集合包中的IntArrayLists来存放偏移和长度。这可以带来两个效果：一是避免整数的装箱和拆箱过程，二是字段保存为原始字节数组的引用从而避免复制。

上述的文件数据解析基于多个很强的假设，这些假设依赖于存在某些限制的输入数据。第一，假设ByteBuffer总有足够的数据来完成当前行。第二，数据假定采用Unix类型的行分隔符，即只有一个换行符而不包含回车符。第三，假设只使用ASCII字符子集。

上述假设使得在实现时可以相当自由，从而读入和解析数据的时间不到基于String代码实现同样操作所花费的时间的1/3。Fastline很好，但却非全部。

2. 直接的数值编码器的接口

数值可以采用多种方式编码成向量。对于连续变量，可以使用ContinuousValueEncoder并将数值以字符串的方式传入，这种做法在前面章节中可以看到。另一方面，如果已经以浮点形式得到了想要的值，那么就可以输入一个NULL字符串和以权重方式表示的权重。

在一个直接的字节解析器中，可以快速访问字段值而不需要将其先转换成字符串然后转换成浮点表示。这可以使得ConstantValueEncoder中的权重字段使用非常具有吸引力。下面的代码给出了上述过程。

代码清单16-3 直接的数值编码

```
public static void main(String[] args) throws IOException {
  FeatureVectorEncoder[] encoder = new FeatureVectorEncoder[FIELDS];
  for (int i = 0; i < FIELDS; i++) {
    encoder[i] = new ConstantValueEncoder("v" + i);
  }
  long t0 = System.currentTimeMillis();
  Vector v = new DenseVector(1000);
  ByteBuffer buf=ByteBuffer.wrap(
      FileUtils.readFileToByteArray(new File(args[1])));
  FastLine line = FastLine.read(buf);
  while (line != null) {
```

```
    v.assign(0);
    for (int i = 0; i < FIELDS; i++) {
      encoder[i].addToVector((byte[]) null,          使用NULL字符串值
          line.getDouble(i), v);
    }
    line = FastLine.read(buf);
  }
  System.out.printf("\nElapsed time = %.3f s\n",
      (System.currentTimeMillis() - t0) / 1000.0);
}
```

上述代码除了使用Fastline而不是Line之外，其他部分和基于字符串的编码程序很类似。这里构建了相同的编码器，仍然一行行地读入数据，但是来自的是Fastline而不是字符串水平的输入。这里对每个字段的编码与前面有少许不同，这是因为FastLine类将字节表示转换成数字来避免编码器来完成这一任务。

上面利用Fastline的版本大概比基于字符串的版本要快5倍，前面已经提到过这一点，但是这种直接数值编码技巧的效果单独拿出来看甚至更加令人印象深刻。在处理百万行数据时，忽略数据解析的时间，从字节表示中对所有数值直接编码需要4~5秒的时间，但从字符串格式来编码则需要40多秒钟。

上述所有加速方式的整体效果是，不需要费多少周折，就可以对原始数据实现15 MBit/s的读取、解析和编码速度。通过使用多线程读取方式，整个转换率可以很容易地与多主轴磁盘传输的速度相匹配。如果采用包含多个交互变量的更精细的编码过程，整个编码开销将会有所上升，但是上述改进方式仍然具有很大影响。

如果在运行大型程序时，上述优化是值得的，特别是直接使用低层CrossFoldLearner或OnlineLogisticRegression对象而没有AdaptiveLogisticRegression介入（mediation）时更是如此，这是因为这些低层的学习器本身很快因此读入训练数据可能是限制性能的因素。另一方面，使用更抽象的基于字符串的方法可以使得代码的编写和调试更加容易，并且在数据比较怪异时不易受到意外。

一旦（快速）读取并（快速）转换了数据，下一个性能瓶颈可能是将分类器集成到服务器这个过程。

16.4 集成 Mahout 分类器

Mahout分类器的集成通常就是一个简单的建立网络化服务的过程。因此，通常网络化服务所要考虑的吞吐率、延迟和服务更新等问题同样要在这里加以考虑。由于Mahout分类器通常会集成到需要很高速度的应用当中，所以这些问题在这里考虑时稍有不同。但同时在某种程度上说，相对于大部分服务而言，基于Mahout分类器的服务相对要简单一些，这是因为它们是无状态的。这种无状态性允许琐碎的水平扩展。

这一节将介绍如何计划并实际将Mahout分类器集成到一个服务架构中。然后，在16.5节中将把这些思想综合在一起用到一个实际运行的分类服务器例子中。

16.4.1 提前计划：集成中的关键问题

尽管基于Mahout的服务具有简单的优点，但是为了充分发挥Mahout的潜力，在构建这种服务之前仍然需要考虑一些关键问题。这些问题包括客户端和服务器端职责的最佳划分方法、检查生产数据和训练数据的不同、速度的设计及模型更新的处理等。

1. 职责分解

将分类模型集成到分类服务中会遇到的一个十分关键的架构方面的选择是系统客户端和服务器端职责的分解。如果分解考虑得十分全面的话，会大大提高系统性能并有助于保障设计能够应对未来的考验。

一般而言，很重要的一点是确保服务器处理特征编码和模型评估。但是，关于在客户端还是服务器端到底各自做多少特征提取，这里存在一个中间地带。例如，如果客户端拥有一个用户ID和一个网页URL，但同时服务器需要从用户资料获取一些信息并且基于网页内容来获得一些内容特征，那么不论是客户端还是服务器端都可以进行必要的与用户资料数据库和网页内容或特征缓冲区的联结操作。如果在客户端进行这些联结和特征提取操作，那么留给服务器端运行的任务将更加有限，这样可能就会得到一个更可靠的服务器。而在服务器端进行上述联结和特征提取操作将使得某些类型的缓冲处理更加容易，并且会对客户端隐藏模型的更多细节。

图16-3给出了如何对客户端和服务器端的职责进行分解的例子。

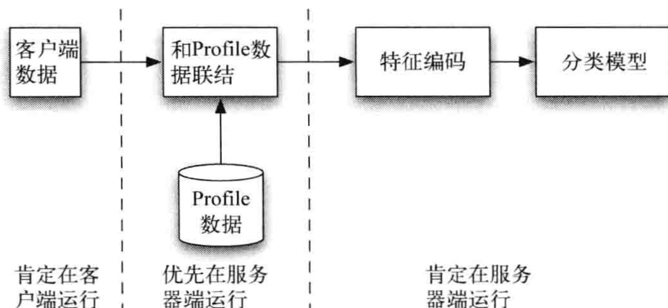


图16-3 和资料数据（如用户资料）的联结操作优先考虑在服务器端运行，但是它们也可以在客户端运行。特征编码应该肯定在分类服务器端运行

一条好的设计原则是，不管客户端程序已经拥有的数据格式如何，模型服务器端都要接受。如果可能的话，除非在模型服务存在之前已经处理，否则服务器不必对客户端已经做的数据准备工作进行进一步的处理。上述原则的一个主要的例外情况是，当允许模型服务器联结资源时，会需要访问那些对客户端已经可用的安全资源。这种情况下，准备特征编码时所需要的联结操作可能不得不在客户端实现。

2. 实时特征是不同的

将模型部署到产品中的一个主要问题是产品中给分类器的数据往往与训练分类器时构造的数据有很大不同。这些不同通常也非有意为之，它们会导致程序出现隐蔽的漏洞，显然，如果对上述问题处理不当会导致很差的分类性能。

那么可能遇到的到底是哪类不同？有些不同来自时间变化过程，这与目标泄漏有些类似。比如，假设想预测用户是否将会购买某个特定商品，并且问题的预测变量基于用户资料的内容。在建立模型的训练数据时，很容易可以得到收集训练数据时而不是购买决策时的用户资料。如果购买过程完成后会有额外信息存储在用户资料中，那么任一使用随时间变化的资料的模型在给定非变化资料时表现很差。

3. 记录分类请求

记录所有的分类请求是检查生产数据和训练数据何时不同的最佳方法之一，甚至在分类系统准备好部署之前就开始记录。记录一小段时间之后，可以使用任意一个自己喜欢的技术来为所讨论的时间段提取出训练数据，然后将提取中的数据与记录的数据进行比较。如果它们不同，训练数据提取过程一定要进行某种程度的改变以避免提取随时间变化的训练样本。最后，可能只有直接记录的数据才能用于训练。

4. 速度上的设计

构建分类系统时速度上的考虑往往会陷入互相矛盾的两个极端。当某个Mahout分类器正确集成时，当然有可能达到极高的速度。如果建立的服务只是对单个输入记录评估单个模型，那么分类服务器不必完成任何大开销的联结操作，模型的评估过程可能会很快导致网络的开销占据主要地位。速度问题将简单归结为寻找一个驱动大量请求通过服务器的访问方法。

例如，不管请求实际需要的时间多短，一个本地运行的Thrift服务器大概需要比50~100微秒略少一点的时间来完成一个服务器请求。由于一个典型模型评估过程仅仅需要大概一千个浮点运算，因此它的完成时间最多不会超过一二十微秒，也就是说90%的时间都花在其他开销上。如果考虑线程和网络传输时间的话，吞吐率会比上面的数字略高，但是很显然大部分时间都不是花在模型评估上。在这种系统中，关键要集中对网络延迟进行优化处理。为达到最高速度，必须要将模型集成到客户端代码中来彻底避免服务器来回处理。这种直接的集成方式使得每个模型评估的延迟最小，但是它可能大幅加剧模型更新的复杂性。

提示 如果需要一次评估一个样本，那么就集中考虑减少网络往返的次数来使得评估延迟最小并考虑客户端模型评估。而对于模型更新来说，可能需要某类服务器来简化模型更新内容的分发而不是依赖于某个可以像本地文件一样访问的文件。

在速度区间的中间段，分类模型的某些用法要求一次评估很多模型，或者允许很多特征向量在单个服务器请求中一起传输。由于在这种批处理评估方法中网络延迟开销可以被多个模型计算过程所分担，因此导致效率的实际提高。相对于每个服务器请求中都包含单个评估的情况，延迟本身并没有降低，但是每次请求中能够完成的计算量大幅度提高。

注意 每次服务器请求中完成大量的模型评估过程是一件好事，特别对于吞吐率来说更是如此。很多目标系统需要对大量对象进行模型评估因此能够允许这种优化处理。

在速度区间的极端段，系统通常需要对极大量的输入进行计算处理。例如，下一章的计算营销系统需要对每个用户计算数十万到数百万的商品，也就说有多少商品就要做多少次模型计算。每个用户提交一个请求，然后使用用户参数、商品参数及用户–商品交互信息来建立一个特征向量。该系统中的模型同时也相当复杂，每件商品都会涉及数百或数千个非零特征。在这种情况下，系统计算的时间可能会达到几百毫秒，此时计算时间主导了整个架构的决策过程。

5. 模型协同更新

对于一个处理分类请求流的生产系统来说，期望100%的正常运行时间相当常见。由于分类器通常是无状态的，通过在负载均衡器后面挂载多个分类服务器构成的服务器池，可以满足上述正常运行时间的需求。每次分类模型更新并发布到生产环境时，上述服务器必须要加载新模型并利用它们来分类。有时候可能还有个额外需求，即要求模型的更新几乎同时对服务器池中的所有服务器可见从而维持服务器间的一致性。

不管确切的需求如何，我们高度推荐使用一个协调服务来管理模型更新并提供活动服务器列表。在这类服务中，Apache ZooKeeper是目前为止最流行的一种，我们强烈推荐使用。Zookepper允许连接一个小型服务器集群，并提供API像访问普通文件系统一样访问集群。除了简单的创建、替换和删除函数之外，ZooKeeper还提供更改通知和一系列一致性保障功能，这些功能能够简化高度可靠的分布式系统的创建过程，甚至在服务器崩溃、维护期和网络分区时也能完成。

基于ZooKeeper建立模型协同更新的架构可以十分简单，特别是不需要精确同步更新时更是如此。在这种架构中，ZooKeeper保留了模型的配置情况。分类服务器会请求配置更改的通知信息，当某个模型正在工作时，这些服务器会维护一个指示文件来告诉分类客户端哪个服务器正在处理请求。而分类客户端通过询问ZooKeeper来确定哪个分类服务器可用。

ZooKeeper中的数据可以按照下列目录结构来组织：

```
/model-farm/
   model-to-serve
   current-servers/
     10.1.5.30
     10.1.5.31
     10.1.5.32
```

在上述目录结构中，/model-farm/model-to-serve文件中包含了所有服务器在用的模型序列化版本的URL地址。每个活动服务器在启动时会读取该文件并在任意更改时让ZooKeeper提供通知。此外，每个服务器每30~60秒会轮询该文件以防通知没有设置到位，这可能是由于服务器启动的时候文件还不存在。

当ZooKeeper通知服务器某个使用模型有更改时，服务器会下载模型的序列化形式并给出模型的一个新实例。然后，该新模型将用于所有后续的请求。一旦新模型正当安装的话，服务器会以唯一的名字在/modelfarm/current-servers目录下建立一个文件。并且要么在文件名上要么在文件内容上，该文件会包含客户端向服务器端发送请求所必需的信息。这些信息可能包含主机名和端口号。当机器拥有多个网络接口时，需要注意确保处理的正确性。

图16-4的梯形图给出了新模型部署的一个典型时间序列。

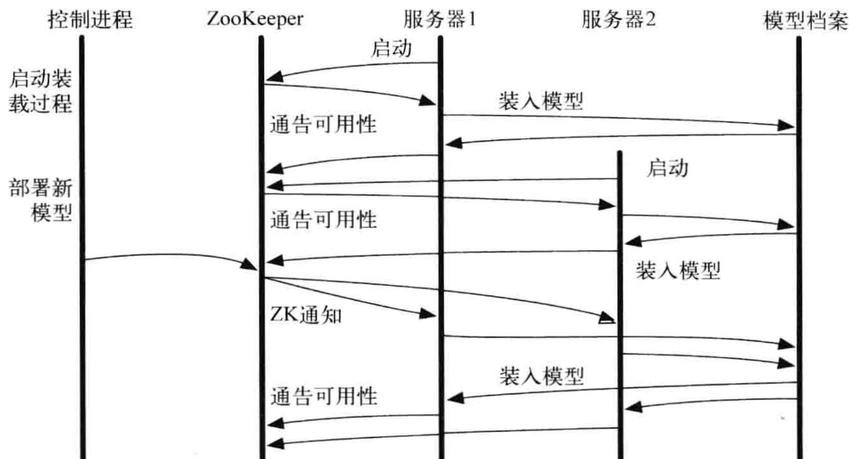


图16-4　如何使用ZooKeeper协调模型部署的示意图。控制进程、服务器1和服务器2之间通过ZooKeeper通信，而当模型修改时ZooKeeper会通知服务器1和服务器2

在图16-4描述的序列中，假定ZooKeeper一开始时带有模型设定值。当服务器1启动时，它会从ZooKeeper装入设定值然后载入模型。当模型全部载入可以提供服务时，服务器1会在ZooKeeper上建立一个文件来表示它已经可用。当服务器2在不久之后启动时，同样会执行上述过程。当控制进程在ZooKeeper上用新模型进行更新时，通知消息会传送给服务器1和服务器2，于是模型下载和可用性通告过程会反复进行。

上述过程理解起来十分简单，实现起来也非常容易。但是，由于所有服务器运行相同的模型并几乎同时更新模型，所以上述过程具有显著的局限性。不难想象，由于每次完成模型装载过程之后服务器就会切换到新模型，上述同时更新会导致整个集群的服务性能下载，而装载过程也可能导致在短时间内不同服务器给出的分类结果不同。类似地，紧接着新模型装入之后，服务器应答可能会有某种程度的延迟，因此所有服务器同时陷入这种低质量服务状态，这可能不是我们所想要的。还有，如果模型文件很大，那么所有服务器同步读取文件可能会因网络开销而造成问题。

下面给出了一个目录结构，展示的是如何对更新进行协调来完成一个更谨慎的模型更新过程：

```
/model-farm/
   should-load/
      node-10.1.5.30
      node-10.1.5.31
      node-10.1.5.32
      node-10.1.5.33
   currently-loaded/
      node-10.1.5.30
      node-10.1.5.31
      node-10.1.5.32
      node-10.1.5.33
   model-to-serve/
      node-10.1.5.30
      node-10.1.5.31
```

```
node-10.1.5.32
node-10.1.5.33
```

在上述目录结构中，顶层的model-farm目录是整个模型集群的名字。在该目录下，should-load目录下包含了以每个服务器地址命名的多个文件，其中每个服务器只对应一个文件。每个文件的内容包含URL和MD5散列列表，这些信息可以用于获取和校验模型的内容。与should-load并列的是currently-loaded目录，它包含多个短期文件，其中每个活动模型服务器构建一个短期文件，文件中包含所有当前该服务器上装入模型的MD5散列列表。最后，另一个与should-load并列的目录是model-to-serve，它包含的是多个文件，其中每个服务器节点对应一个文件，其中包含的是用于所有到来的请求的单个模型的散列值。

在操作中，每个模型服务器会对should-load目录和model-to-serve文件维护一张监视表。should-load目录发生变化表示模型应该装入或卸载，而model-to-serve文件发生变化表示所有后续请求应该导向所指示的模型。每当模型装入或者卸载时，currently-loaded目录下相应的临时文件会更新。

相对于前面的模型，这种目录结构能够在复杂实现的同时允许相当的灵活性。这种灵活性能够允许新模型在基本同步之前就能在多台机器上载入运行。它还能允许只在一台服务器上装入并运行新模型，这样可以在该模型推广到其他所有服务器之前测试它的稳定性。使用MD5散列允许新内容下的URL的重用性，并且可以允许服务器来校验它们是否装入了预期的模型。

实际当中，可能需要将上述模型扩展到控制多个模型服务器农场，或者将模型推广到逐步增多的机器的过程自动化。对于服务器来说更新其状态文件来指示能够处理的流量也是一件很普遍的事情。在选择发送服务器对象时，发送请求的客户端可以使用这些指示信息。

另外，需要记住的是，即使使用内部包含多个模型的`AdaptiveLogisticRegression`来训练模型，也只需要从下面的`CrossFoldLearners`中保存一个模型。这一点非常重要，这是因为`AdaptiveLogisticRegression`包含了大量分类器，而每个分类器内部都潜在包含大量的系数矩阵。如果特征向量很大，那么序列化的`AdaptiveLogisticRegression`可能实际上会有数百兆。

16.4.2 模型序列化

在Mahout 0.4中，SGD和朴素贝叶斯模型的序列化方式不同。SGD模型共享一个称为`ModelSerializer`的辅助类，该辅助类处理所有SGD模型的序列化及反序列化。与此不同，朴素贝叶斯模型只序列化为训练中创建的多个文件的副产品，而朴素贝叶斯模型的反序列化并不基于单个方法来实现，而是显式将上述文件读回到内存中。

在Mahout的未来版本中，有可能将`ModelSerializer`或类似的类扩展为能够同时处理SGD模型和朴素贝叶斯模型。在那之前，朴素贝叶斯模型的序列化和反序列化过程仍然是个微妙而又高度变化的过程。在更可用的序列化接口完成之前，这也意味着朴素贝叶斯模型的部署要使用前面描述的命令行界面而不是以编程的形式来实现。

对SGD模型使用`ModelSerializer`类

`ModelSerializer`类提供了将模型序列化为文件和字符串的静态方法。这个类的使用就像

下面给出的片段所示那样简单，该片段中来自AdaptiveLogisticRegression的某个模型和完整的集成模型分别保存在不同文件中：

```
if (learningAlgorithm.getBest() != null) {
   ModelSerializer.writeBinary("best.model",
      learningAlgorithm.getBest().getPayload().getLearner());
 }
 ModelSerializer.writeBinary("complete.model", learningAlgorithm);
```

所有不同类型的SGD模型都可以使用上述相同的方法来序列化。

从文件中读取一个模型就像下面那样简单：

```
learningAlgorithm = ModelSerializer.readBinary(
              new FileInputStream("complete.model"),
              AdaptiveLogisticRegression.class);

OnlineLogisticRegression bestSubModel = ModelSerializer.readBinary(
        new FileInputStream("best.model"), OnlineLogisticRegression.class);
```

上述片段给出了ModelSerializer的两种用法。第一种用法可能是用于重新装入一个完整的拥有子模型的AdaptiveLogisticRegression类。另一种用户是用于装入单个的OnlineLogisticRegression，其可能是AdaptiveLogisticRegression中组成模型的一个。需要注意的是，要向readBinary提供相应的类，该方法才能知道需要构建和返回的对象类型。

当将SGD模型序列化为分类器进行部署时，通常最好的方法是只序列化AdaptiveLogisticRegression中性能最优的子模型。最终得到的结果序列化文件将会比序列化AdaptiveLogisticRegression对象中包含的整个集成模型所得到的结果小100倍。此外，当有数据需要分类时，由于只有AdaptiveLogisticRegression对象中的最佳子模型被使用，而其他模型都被忽略，因此个体模型更适合部署。

另一方面，如果序列化模型时要求后面还可以继续训练，那么序列化整个AdaptiveLogisticRegression或许是更好的思路。

16.5　案例：一个基于 Thrift 的分类服务器

将一个工作分类服务器的各部分拼成一起可能是件令人生畏的事情。为了帮助实现这一点，本节会给出一个完整的工作样例，该样例中会展示所有我们已经讨论过的东西。我们已经编写了一个简化但很完整的分类服务器，该服务器实现了16.4节中介绍过的那个更简单的部署方案。该服务器提供一个全功能负载均衡分类器所需要的全部基本功能，也包括一些管理功能。

该例子中分类客户端和服务器端的通信使用Apache的Thrift（http://thrift.apache.org）来处理。Thrift是Apache的一个可以以十分简单的方式构建客户端-服务器端应用的项目。

本例当中多个服务器及服务器和客户端之间的协调工作使用Apache的ZooKeeper（http://thrift.apache.org）来处理。本例中，ZooKeeper持有所有分类服务器有关装入哪个模型的指令，还保存了所有服务器的状态信息，客户端通过这些状态信息了解哪台服务器已经结束、每台服务器上运行哪个模型。这种服务器协调的结构参见图16-5。

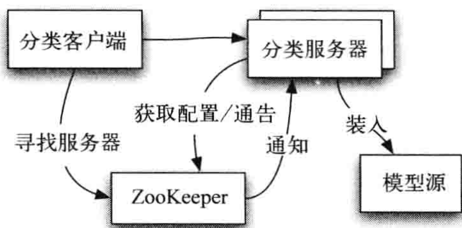


图16-5 ZooKeeper作为分类客户端和服务器端的协调和通知服务器

这种安排可以使得分类服务器知道模型定义的位置，客户端也可以找到所有的活动服务器。

当某台服务器启动时，它会询问ZooKeeper来找到装入的模型。装入该模型后，它就会向ZooKeeper写入一个文件来通告所有机器它可以接受流量。每当应该装入哪个文件的指示文件改变时，ZooKeeper会将该更改通知所有的服务器。

当客户端希望向服务器发送一个查询时，它会浏览ZooKeeper来找到当前正在提供服务的服务器并从中随机选出一台服务器。

用于协调的ZooKeeper中的多个文件的结构如下：

```
/model-service/
   model-to-serve
   current-servers/
     hostname-1
     hostname-2
```

model-to-serve文件给出的是被所有服务器装入用于分类的模型的URL。这些服务器将维护该文件的一个监视表从而一有修改就能得到通知。当它们得到修改通知时，会装入model-to-serve文件来获得新模型的URL然后重新装入模型。服务器一装入任一模型，就会在current-servers目录下建立一个以服务器名字命名的临时文件。由于该文件是临时的，所以如果相应服务器崩溃或者退出，该文件就会在数秒之内消失。

上述模型服务器的主类代码如代码清单16-4所示。其主要包含Thrift服务器层的创建代码，但是需要注意的是ZooKeeper中如何使用一个计时器来按计划进行周期检查。这种做法可能完全冗余，但是这是“吊带检查式”（belt-and-suspender）编程风格 的一个很好的例子。ZooKeeper应该会通知任意的修改，但是时常检查可以确保不错过由于编程错误导致的某个故障。

代码清单16-4 分类服务器的主程序

```
public static final String ZK_BASE = "/model-service";
public static final String ZK_CURRENT_SERVERS =
               ZK_BASE + "/current-servers";
public static final String ZK_MODEL = ZK_BASE + "/model-to-serve";
private final TServer server;
private final Logger log =
               LoggerFactory.getLogger(this.getClass());
private final ZooKeeper zk;
private final Ops modelHandler;
private Timer timer;
```

```
private String currentUrl = null;
private int version;
private Watcher modelWatcher = new Watcher() {           ❶ 代码清单16-5中会给出细节
  ...
}
public Server(int port)
      throws TTransportException, IOException,
                    InterruptedException, KeeperException {      连接本地ZooKeeper
  zk = new ZooKeeper("localhost", 2181, null);
  modelHandler = new Ops();                                      建立分类器内部的服务
  timer = new Timer();                          每30秒钟重试模型的装入过程
  timer.scheduleAtFixedRate(new TimerTask() {
  @Override
  public void run() {
   modelWatcher.process(null);
  }
}, 0, 30000);                                                  建立Thrift服务器
socket = new TServerSocket(port);
Classifier.Processor processor = new Classifier.Processor(modelHandler);
TProtocolFactory protocol = new TBinaryProtocol.Factory(true, true);
server = new TThreadPoolServer(
    new TThreadPoolServer.Args(socket).processor(processor));
log.warn("Starting server on port {}", port);

  server.serve();                               启动服务器
}
public static void main(String[] args) throws IOException,
               TtransportException, InterruptedException, KeeperException {

  new Server(7908);
}
```

大部分行为都在`modelWatcher`对象❶中。该对象是ZooKeeper `Watcher`的一个实现，每当model-to-serve文件发生修改时就会触发对该对象的调用。下面给出的是`modelWatcher`的源码。

代码清单16-5　装入模型并设置模型状态的`Watcher`对象

```
private Watcher modelWatcher = new Watcher() {
  @Override
  public void process(WatchedEvent watchedEvent) {
    String hostname = null;
    try {
   hostname = InetAddress.getLocalHost().getHostName();
 } catch (UnknownHostException e) {
    }
 if (hostname == null) {
  log.error("Must have hostname ... exiting");
      System.exit(1);
 }
 String url = null;
 try {                                              ❶ 从ZooKeeper中获得URL
   Stat stat = new Stat();
   byte[] urlAsBytes = zk.getData(ZK_MODEL, modelWatcher, stat);
   int latestVersion = stat.getVersion();
```

```
    url = new String(urlAsBytes, Charsets.UTF_8);
    if (currentUrl == null || latestVersion != version) {        <-- 检查URL是否修改

      URL modelUrl = new URL(url);
      log.warn("Loading model from " + modelUrl);
      AbstractVectorClassifier model = ModelSerializer.readBinary(
          modelUrl.openStream(),
          OnlineLogisticRegression.class);
     try {                                                        ❷ 通知ZooKeeper模型
        zk.create(ZK_CURRENT_SERVERS + "/" + hostname,         <-- 服务器已经装入
                    modelUrl.toString().getBytes(Charsets.UTF_8),
                    ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL);
      } catch (KeeperException.NodeExistsException e) {

        zk.setData(ZK_CURRENT_SERVERS + "/" + hostname,              <--
                    modelUrl.toString().getBytes(Charsets.UTF_8), -1);
      } catch (KeeperException e) {
        log.error("Couldn't write server status file");       更新先前存在的文件
      }                                                                 ❸
      modelHandler.setModel(model);
      currentUrl = url;
      version = latestVersion;
      log.info("done loading version " + version);
    }
     return;                                                   如果ZooKeeper中没
  } catch (KeeperException.NoNodeException e) {              <-- 有数据，则抛出错误
    log.error("Could not find model URL in ZK file: " + ZK_MODEL, e);
    return;
  } catch (KeeperException e) {
    log.error("Failed to load model due to ZK exception", e);
  } catch (InterruptedException e) {
    log.error("Operation interrupted should never happen", e);
  } catch (IOException e) {
    log.error("Failed to load model from " + url, e);
  }
  log.warn("Clearing current URL due to error");
  currentUrl = null;
  version = -1;
    }
  };
```

上述代码中不少都用于处理异常情况，但是基本的框架很简单。代码中的骨干部分在`zk.getData`❶、`zk.create`❷、`zk.setData`❸的调用中。

上述代码运行如下：它尝试从ZooKeeper读取模型的URL。如果URL读取之后并不存在当前模型，或者ZooKeeper文件包含的URL版本和当前模型不一样，那么模型都会重新装入并且该服务器的状态文件会采用此模型URL来更新。更新时，首先尝试创建状态文件，如果创建失败则更新状态文件。

如果由于模型文件丢失导致URL的原始读取失败的话，那么就会记录下一条消息然后发生一个相对正常的返回过程。其他造成服务器当前模型URL失效的错误情况下，要确保下一次重新装入模型。

实际的分类请求处理由Thrift服务器来完成。Thrift IDL中定义的接口如下所示：

```
namespace java com.tdunning.ch16.generated
service Classifier {
  list<double> classify(1: string text)
}
```

上述接口定义声明服务器会接受分类请求。每个请求将会返回一个得分列表，每个得分反映的是每个可能目标变量值的可能性。

前面IDL定义的接口的实现在Ops类中，具体代码如下。

代码清单16-6　分类服务的实现

```
public class Ops implements Classifier.Iface {
  private static final int FEATURES = 10000;

  private static final FeatureVectorEncoder enc =
         new TextValueEncoder("text");
  volatile AbstractVectorClassifier model;

  public Ops() {}

  @Override
  public List<Double> classify(String text) throws TException {
    Vector features = new RandomAccessSparseVector(FEATURES);
    enc.addToVector(text, features);

    Vector r = model.classify(features);                    <-- 对文本分类

    List<Double> rx = Lists.newArrayList();
    for (int i = 0; i < r.size(); i++) {                <-- 转换成输出格式
      rx.add(r.get(i));
    }
    return rx;
  }

  public void setModel(AbstractVectorClassifier model) {   <-- 为ZooKeeper监视器
    this.model = model;                                      提供钩子（hook）❶
  }
}
```

classify方法使用当前装入的模型对特征向量进行分类。当Thrift服务器收到一个客户端请求时，就会调用上述方法来将文本按照第15章TrainNewsGroups程序的风格编码后传给它。然后就使用模型计算并返回一个得分向量。

Ops类中也定义了一个setModel方法❶，ZooKeeper监视器类可以利用该方法在观察到模型更改时装入新模型。

16.5.1　运行分类服务器

为观察上述服务器的实际效果，可以使用ZooKeeper的命令行界面来触发其行为。然而在开始之前，必须要有一个可工作的模型。代码清单15-3中的TrainNewsGroups就是一个十分便利的获取上述模型的程序，因为它每几百个训练样本就将一个模型副本写入/tmp中。

建立和运行TrainNewsGroups的过程在源代码附带的README文件中进行描述。

当运行TrainNewsGroups程序时，输出量十分巨大。输出中的一些关键片段如下所示：

```
[INFO] [exec:java {execution: default-cli}]
11314 training files
0.62 189992.00 ... 7000  -1.017 81.02 none
0.64 189992.00 ... 8000  -0.898 84.77 none
0.75 189997.00 ... 10000 -0.948 84.71 none
...
```

同时，可以得到/tmp下保存的模型文件：

```
$ ls -l /tmp/
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1000.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1200.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1400.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1500.model
...
-rw-r--r--  1 ...  1680247 Nov 21 01:19 news-group.model
$
```

上述名如news-group-*.model之类的文件包含学习每一步中的最佳模型。

一旦拥有一些模型文件，就可以像第16章源代码附带的README文件中的说明描述的那样启动ZooKeeper的一个本地副本。

ZooKeeper运行后，可以启动分类服务器：

```
...
10/11/21 01:36:06 INFO zookeeper.ClientCnxn: Socket connection established to
     localhost/fe80:0:0:0:0:0:0:1%1:2181, initiating session

10/11/21 01:36:06 WARN ch16.Server: Starting server on port 7908
...
10/11/21 01:36:07 ERROR ch16.Server: Could not find model URL in ZK file: /
     model-service/model-to-serve
10/11/21 01:36:08 WARN ch16.Server: Starting server on port 7908
...
```

上述过程最好使用与启动ZooKeeper不一样的另一个窗口来完成，这样当日志行出现时才能对它们进行分离。

迄今为止，分类服务器已经运行，但是ZooKeeper中没有关于装入并使用哪个模型来对请求进行分类的任何信息。为将正确信息放入ZooKeeper，可以使用下面所示的ZooKeeper命令行界面。同样，为分离信息也最好使用一个新窗口。

```
$ ~/Apache/zookeeper/bin/zkCli.sh  <<EOF
create /model-service ""
create /model-service/model-to-serve \
          file://localhost/tmp/news-group-1500.model
quit
EOF
```

这些命令应该会产生大量输出，最重要的输出行可能如下所示：

```
[zk: 1] Created /model-service
[zk: 3] Created /model-service/model-to-serve
```

16

在用户与ZooKeeper交互的同时，前面启动的分类服务器每30秒都检查一次看看ZooKeeper是否正确建立。每次检查时，可能都会发出同样的不能找到模型URL的错误消息。但是，在最终建立/model-service/model-to-serve文件后的循环中，可能会产生一个不同的消息：

```
10/11/21 01:45:04 INFO ch16.Server: Loading model from file://localhost/tmp/
    news-group-1500.model
10/11/21 01:45:04 INFO ch16.Server: model loaded
```

这时候，分类服务器正常运行。可以通过修改model-to-serve文件来看看ZooKeeper配置命令的反应：

```
~/Apache/zookeeper/bin/zkCli.sh  <<EOF
set /model-service/model-to-serve file://localhost/tmp/xxx.model \
quit
EOF
```

几乎在刚刚输入上述命令之时，会发现服务器输出不能发现刚才假造的模型的警告。可以利用下面的命令回到正确状态：

```
~/Apache/zookeeper/bin/zkCli.sh  <<EOF
set /model-service/model-to-serve \
      file://localhost/tmp/news-group-1500.model
quit
EOF
```

也几乎在同时，分类服务器将会确认已经装入模型。如果有10个模型服务器运行在不同的机器上，它们几乎都会那么快就有响应。

16.5.2　访问分类器服务

在客户端，事情甚至还要简单。下面的代码清单给出了一个客户端如何从ZooKeeper获取活动服务器信息然后向其中一个服务器发送分类请求的过程。

代码清单16-7　访问分类服务器

```
public class Client {
  public static void main(String[] args) throws TException, IOException,
       InterruptedException, KeeperException {
                                                           ❶ 获取活动服务器列表
     ZooKeeper zk = new ZooKeeper("localhost", 2181, null);  <┘
     List<String> servers =
             zk.getChildren(Server.ZK_CURRENT_SERVERS, false, null);
     if (servers.size() == 0) {
       throw new IllegalStateException("No servers to query");
     }
                                                         ❷ 随机选择服务器
     int n = new Random().nextInt(servers.size());      <┘
     String hostname = servers.get(n);
     Connection c = new Connection(hostname, 7908);           <┐
     List<Double> result1 = c.classify("this is some text to  发送请求
    classify");                                                并打印结果
     System.out.printf("%s\n", result1);                         ❸
     List<Double> result2 = c.classify("Given that the escrow " +
            "keys are generated 200 at a time on floppies, why\n" +
```

```
                 "not keep them there rather than creating one huge" +
                 " database that will have to\n" +
                 "be guarded better than Fort Knox.");
      System.out.printf("%s\n", result2);
      c.close();
    }
  }
```

客户端做的第一件事是联系ZooKeeper来获的活动的服务器列表❶并从中随机选择一个❷。之后，它连接那个随机选中的服务器并请求其对两段文本进行分类❸。

客户端程序的输出结果应该看上去如下面所示，其中为了清晰起见忽略了某些片段。

```
$ mvn exec:java -Dexec.mainClass="mia.classifier.ch16.Client"
...
[INFO] Preparing exec:java
...
10/11/21 20:15:26 INFO zookeeper.ClientCnxn: Opening socket connection to
     server localhost/0:0:0:0:0:0:0:1:2181
...
[0.03964640884739735, 0.07565438894170683,
0.05901603385987076, 0.05114326116258236,
0.050421016895119145, 0.044210065098318506,
0.04114422304836256, 0.05402402583820334,
0.05281506587810628, 0.042271194810117395,
0.07788774620593823, 0.048761793702438536,
0.03998325394587257, 0.03846382850793468,
0.04373069260414459, 0.04980658400349993,
0.04722614673439341, 0.04948828603244314,
0.04477719025270829]                                        第一个请求

[0.030532890514017766, 0.25984369795811524,
0.03596784998912587, 0.05321219256232682,
0.02395166487009193, 0.033935125988576176,
0.03729692334981463, 0.04035766767461885,
0.05067830734673832, 0.03391626207189733,
0.07418249664999789, 0.06666407060825316,
0.0424634477415305, 0.01074681122380413,
0.05212145428368258, 0.030600556819694907,
0.025032370110624636, 0.02760955132545441,
0.02696366801467259]                                        第二个请求
[INFO]
 ------------------------------------------------------------------------
[INFO]
 BUILD SUCCESSFUL
...
$
```

上述两个请求的输出结果为一系列的数字列表，其中每个数字都是某个不同新闻组的得分。由于第二个请求抽自真实的新闻组帖子，因此它更有意思一些。该请求也在第二个新闻组那儿得到一个显著的高分值，如你所想的那样，当给定分类器数据进行分类时就会得到一个结果。

上述例子中编写的客户端内在地就在多个服务器之间进行了负载均衡处理。此外，一个稍微先进一点的顺序执行多个查询的客户端可能会在ZooKeeper的current-servers目录下维持一张监视

表。一旦该监视表被触发，客户端可以选择另一个服务器。只要任一新服务器上线，或者某个活动服务器退出或崩溃不久，上述做法就能保持立刻的负载均衡。

上述基于Apache Thrift的服务器案例还不足以正式使用，但是可以看成很多类实际产品级部署的一个基本骨架。上述系统设计中的一个关键环节是在多服务器环境下使用ZooKeeper在分类客户端和服务器端之间进行协调和通知。这种做法能够使得分类服务器快速透明扩展，并且能在任意时刻更改活动模型。

16.6 小结

到这里要对读者表示祝贺！因为您已经成功操作了Mahout分类中的第三步也是最后一步，即部署一个大规模的分类器。迄今为止，读者知道了如何为分类系统构建和训练模型、如何评估和对模型进行微调来提高性能、如何在巨型系统中部署训练好的模型等。从本章学习到的有关部署的关键性结论包括完整理解整个项目、设计时对系统中任意可能超出Mahout容易做到的部分要特别注意。提前发现这些潜在的困难可以有机会修改设计来避免问题的出现或者安排时间来拓展边界。

通常情况下，部署一个大型系统的最紧要的部分是建立训练流水线。在需要Mahout处理的规模的系统中，上述流水线可能会带来一些实质性的工程挑战问题。成功处理的一个关键是，在进行极大规模联结操作时采用并行方式实现流水线，关于这一点前面已经学过。

本章也给出了一些需要避免的陷阱，如避免生成目标泄漏或造成语义失配。我们给出了这些漏洞可能出现的多种隐蔽方式并给出了避免办法。

最后一个学到的经验涉及服务的设计。本章给出了一个基于标准技术（如ZooKeeper和Thrift）的健壮的分类服务的核心设计案例。如果理解了该服务设计的精神，那么就可以可靠地反复部署高性能的分类服务。但是千万不要被本章设计的简洁性所欺骗。系统的大部分简洁性、健壮性和可靠性都建立在ZooKeeper坚实的基础上。如果远离这里的基本设计思路的话则要三思而后行。

正如读者看到的那样，当集成到大型系统中，即使部署一个设计良好的分类器都是一件复杂的事情。下一章我们要结束对分类的探讨，其中将会给出将上面所讲的东西都集中在一起的真实世界的一个大规模分类器系统案例，该系统为一个在线贸易公司Shop It To Me所使用。

第 17 章

案例分析——Shop It To Me

本章内容

- 分类系统速度和扩展性方面所考虑的问题
- 构建训练流水线的过程
- 极高吞吐率下的分类器重构过程

迄今为止，前面几章内容已经给出了有关分类的一个总体介绍，除此之外，还包括设计和训练Mahout分类器的详细解释、对训练模型进行评估以将性能调整到想要的水平以及将分类器部署到大型系统等内容。本章将通过一个实际案例将上述所有主题付诸实践，该案例来自一个真实的在线贸易公司Shop It To Me（http://www.shopittome.com），该公司选择Mahout作为其分类方法。我们将会看到该公司的一个小的工程团队在建立和部署高性能Mahout分类器时遇到的问题和解决的办法。

第16章主要关注超大系统的规模需求，该需求最好通过Mahout分类来完成。类似地，本章案例也处理大规模数据集，但同时它也提供了一个即使对Mahout系统来说速度需求都很极端的例子。出于扩展性特别是速度上的要求，开发团队所涉及的解决方案中需要大幅度的实质性创新。总的来说，本章介绍的系统将会展示 Mahout分类器比初看起来更强大的一面。

在介绍Shop It To Me系统之后，会考察其外发邮件系统的工作过程并介绍该系统产生的数据。该案例分类器的目标会随着模型训练的每一步来介绍。然后考察Shop It To Me团队是如何使得训练和模型计算的过程足够快来满足业务需求的。

最后，我们总结本案例中学到的经验以便能够将学到的知识用到自己的项目中去。

17.1 Shop It To Me 选择 Mahout 的原因

Shop It To Me这个案例会揭示为什么Mahout对于某些规模和速度的问题是十分理想的选择方案，而这些问题的其他方案却不能很好解决。但是为了让读者深刻理解该案例中面对的挑战，了解一下公司的背景信息十分有益。接下来，我们看看Shop It To Me公司是干什么的、他们需要分类系统干什么以及为什么选择Mahout来构建该系统。

17.1.1　Shop It To Me公司简介

Shop It To Me是一个免费的在线个人购物服务，它通过邮件（SaleMail）通知为顾客提供感兴趣的销售商品。公司的使命是及时在顾客和满足顾客偏好的销售商品之间建立联系。由于用户交易往往与临时打折的商品有关，因此及时性特别重要。如果需点击的商品链接没有及时传送给顾客，该商品将会在购买之前消失。简而言之，公司的使命就是展示顾客想要看的商品。

Shop It To Me与几百个零售商之间有合作关系，因此公司能够知道商品上市销售的时间。迄今为止，Shop It To Me有超过300万的顾客，为构建发送给顾客的邮件（即SaleMail），公司每个月会给出超过20亿的商品推荐信息。

17.1.2　Shop It To Me需要分类系统的原因

读者可能会有疑问，上述商业使命本质上基于推荐形式，即预测特定顾客喜欢哪些商品，这种需求为什么需要分类呢？这里选择分类的原因在于在Shop It To Me这种情况下常规的推荐方法不太理想。在商品被很多用户看过之后推荐系统会表现很好。经典的例子包括电影和音乐的推荐。通过对某些相同商品的交互相似度可以找到用户之间的关联，这种相似度可以预测用户的其他偏好。最重要的一点是，推荐系统中被推荐的商品第二天仍然存在。

但是，Shop It To Me推荐给顾客的商品并不存在这种持久性。不夸张地说，最畅销商品即使今天可能还在但是明天可能就断货。商品的这种短暂性本质意味着推荐系统必须要考虑训练数据中商品的特性，比如品牌、价格、颜色等，这些信息在后来的商品中也会存在。为实现这一点需要另一种方法，即作为一个类推荐系统的一部分，利用分类来进行预测。在Shop It To Me，可用的数据包括用户的个人历史以及要推荐商品的特性信息。这些信息构成了所有的预测变量。这里也有一个很好的目标变量，即用户是否会点击某件商品。

为了提高这些推荐的作用，Shop It To Me构建了一个先进的分类系统提供来帮助精确预测哪些商品会吸引特定的用户。当为某个特定用户构建一封SaleMail时，该分类系统会计算已训练分类模型的值来预测对于数十万可能的商品中的每一个用户是否会点击。通过按照预测的用户偏好概率对商品排序，Shop It To Me可以构造特定兴趣的SailMail发给用户。

17.1.3　对Mahout向外扩展

构建上述系统会面对一些特殊的挑战。不仅要面对几十亿的训练样本规模，还要达到惊人的分类速度。粗略计算一下，数百万用户需要对几十万商品进行模型评估，也就是说有几千亿的模型评估。为满足邮件构建和传输流水线的约束条件，上述分类过程必须要在几小时内完成。也就是说，必须要在每秒内完成数千万次分类，这个数字按照我们老家的说法叫“a lot”。[①]

为在合理的计算构架下满足上述苛刻的要求，Shop It To Me已经不得不做了一些非常有趣的工程工作。多个系统用于产生原型分类器，包括R（http://www.r-project.org/）和Vowpal Wabbit

① 第一作者Sean Owen来自英国伦敦。——译者注

（https://github.com/JohnLangford/vowpal_wabbit/wiki），后者是雅虎资助的一个开源SGD分类器。利用R工具对训练大数据量进行简单处理并同Shop It To Me已有的Ruby/JRuby架构进行良好集成被证明是十分困难的。而Vowpal Wabbit能够处理训练数据的规模，但是将Vowpal Wabbit很好集成到已有架构也十分困难。

另一方面，Mahout可以处理训练规模并且可以和已有架构很好地集成。在Shop It To Me系统中， Mahout提供了关键的部分，这些部分将在本章当中详细介绍。Mahout不一定是任一项目的优先选择，但是它却被证明是面对Shop It To Me中工程挑战的最佳选择。

注意 Shop It To Me系统设计、关键变量提取及模型性能的一些细节属于公司机密，这里予以省略或者略过。

系统的整个框架在这里只是泛泛地描述，并不包含Shop It To Me系统各方面的精确细节。但是，为理解扩展问题及其解决方案给出了足够的细节。这些问题随着邮件系统本身的结构而产生。

在 Ruby 中使用 Mahout

Shop It To Me 是一个基于 Ruby 的商店，但是它选择了基于 Java 的 Mahout 来作为建模的平台。这看上去有点自相矛盾，但是正如 Shop It To Me 的工程师确定的那样，Java 非常容易集成到一个基于 Ruby 的 Web 服务和进程中去。其中一个主要原因在于 JRuby 允许从 Ruby 代码中调用 Java，这种调用几乎是透明的，而且和标准 Ruby 相比性能并没有显著降低。这使得可以使用标准的 Ruby 风格和工程实践方法来构建分类服务。

另一方面，Shop It To Me 的模型训练使用 Cascading 集成到 Rake 定义的 Ruby 工作流中实现。而通过某种领域专用语言（DSL）撰写 Cascading 工作流在系统中隐藏了 Cascading 的 Java 本质。

尽管扩展或维护现有系统仍然需要重要的 Java 专业知识，Mahout 和 JRuby 之间的大部分集成并不需要丰富的 Java 知识，而只需要标准的 Ruby 方法和工具就可以完成。

17.2 邮件交易系统的一般结构

Shop It To Me的整体数据流大致如图17-1中的路径所示。该图给出了如何利用注册信息来填充用户表格和品牌偏好表格。该图也展示了如何使用利用专门的数据导入器从零售伙伴中发现销售商品。系统的核心部件是SaleMail构建器，它会访问包含当前销售商品、用户及其兴趣信息的表格。包含一个点击预测模型在内的模型用于选择在发给每个用户的邮件中选择哪件商品。每封邮件中包含的商品记录在一个商品出现表中。

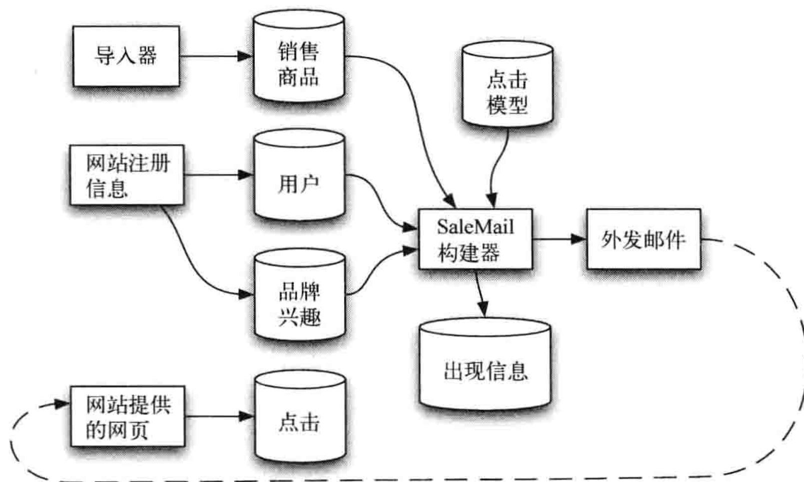


图17-1　Shop It To Me邮件销售系统的总体数据流图。虚线给出的是用户通过点击邮件中的商品而提供反馈的过程

如图17-1所示的那样，用户及出售的商品构成了邮件生成器使用的点击模型所需的预测变量数据。当用户点击其收到邮件中的商品时，会导致网页的访问，这些访问会被日志文件所记录。然后，这些记录到的点击信息会放到训练数据中用于下一轮的建模，这样点击模型就可以更新。这些数据集中的一部分被存在关系数据库中，比如用户表，但是其他一些数据集，比如出现表可以存在大规模日志文件集合或进行隐式存储，一旦需要就必须重构。

Shop It To Me系统产生的一封邮件示意图如图17-2所示。我们会看到邮件的上部提供了多种腕表以供选择，邮件的下部还给出了更多的商品。



图17-2　从Shop It To Me发送的推荐邮件中截取的一段内容

图17-1工作流中用到的数据库存放了整个系统的状态信息，包括可购买的商品、已经发送的邮件以及用户表示出的偏好等信息。图17-3给出了Shop It To Me系统中用到的表结构的一个简化版本，这些表结构用于记录构建点击模型所需的用户、商品以及它们之间的交互行为信息。

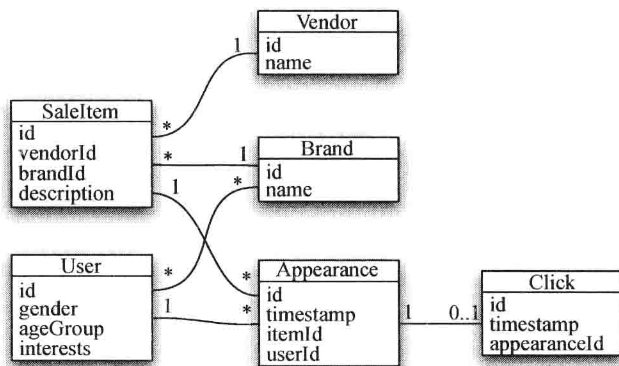


图17-3 支持Shop It To Me数据流的数据库表的简化UML视图。每件销售商品包含一个唯一的供应商和品牌，但是可以出现多次。同样，用户可能表现出对多个品牌的兴趣，但是会浏览很多商品。每次出现可能导致至少一次点击行为

在图17-3中，User和SaleItem表的非规范化视图会导出为Apache Hadoop文件用于构建训练数据。由于规模原因，Appearance表并不是真正的数据库表。实际上它保存在传统文件中，文件中包含发送的邮件记录。Appearance表和Click表都会导出到Hadoop来联结。需要注意的是，如何以高度规范化的格式来保存这些表，这对于处理事务和支持Web访问来说相当重要，但是这种规范化在训练时必须要去掉。

迄今为止，我们已经了解了Shop It To Me邮件系统的框架以及销售过程的组织方式。想象一下如何设计分类系统。不管要建立何种分类器，Shop It To Me第一步要做的事情是训练模型。

17.3 训练模型

训练一个模型首先需要一些初步规划。必须要考虑如何提出问题来满足目标的需要并确定系统输出中的目标变量。必须要考察可用的历史数据，以了解哪些特征可以用作变量，确定其中的哪些特征可以作为最有效的预测变量。对于Mahout分类器来说，要审视所做项目的关键点特别重要，这样才能以最佳方式确定如何在速度和规模需求之间寻求平衡，我们在前面章节中已经讨论过这一点。

本节将介绍Shop It To Me如何完成上述步骤。

17.3.1 定义分类项目的目标

Shop It To Me的工程师们很像我们在前面章节提到的那样解决他们的问题。其分类项目的目

标是，基于商品的出现信息和用户的点击历史构成的训练数据来预测用户是否会点击该商品。点击预测是目标，而用户和商品的特性是预测变量。然而，要达到可用状态，Shop It To Me数据需要大量的预处理工作。

训练数据的建立流程如图17-4所示。该图给出了用户数据的预处理及其与商品数据的联结过程，它们一起构成了模型训练所要的数据。

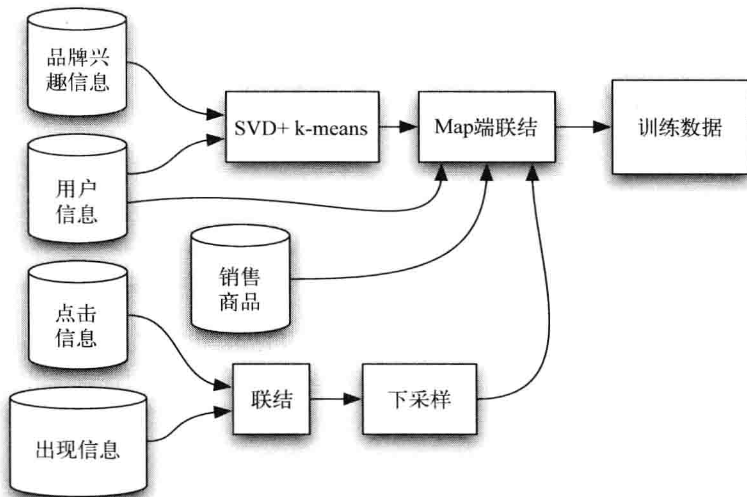


图17-4 模型训练数据建立中的数据流图。点击和出现信息以并行的方式联结并进行选择性下采样处理。然后所有的数据联结在一起产生训练数据

Shop It To Me系统中拥有的一个最丰富的信息是品牌偏好数据库。当用户注册到Shop It To Me系统时，会回答系统有关品牌偏好的问题。他们的回答存储在品牌偏好数据库中，这是一个非常优质的关于用户偏好的信息源。不幸的是，该数据的格式并不能直接为分类所用。

为解决上述问题，首先使用SVD分解来对数据进行约简然后采用k-means聚类算法进行聚类。SVD分解是一种常常用于对观察数据降维的数学技术。上述处理之后，表现出相似品牌偏好的用户会被分到同一个簇中。品牌偏好簇的ID以某种十分适合于建模的格式表示了原始品牌信息。

在全部计算当中，Appearances表是最大的一张表，最终训练数据当中的每条记录都可以回溯到该表中的唯一一行。此外，根据商品的出现是否导致一次点击对Appearances表按照不同采样率进行了下采样。未点击的出现信息基本上都会进行下采样，但是点击的信息不参与下采样。为使下游的联结操作处理更少的数据因而更加高效，点击和出现信息的联结操作在一个完整的reduce联结中完成，并且在与用户和商品元数据联结之前在Reducer中进行下采样处理。

然后，点击及出现数据和用户元数据及商品元数据进行联结操作，其中用户元数据包括性别、年龄、大体的地理位置及品牌偏好簇等，而商品元数据包括商品编号、颜色、尺寸和描述信息等。这里的联结操作将利用一个Map端的联结来完成，这是因为用户和商品元数据表只包含几百万记录，放到内存相对容易。

上述联结也可以嵌入一个Reduce端的联结运算并集成到点击和出现信息的联结步骤中完成。通过尝试更多实验，有可能提高上述实现方法的性能。

17.3.2 按时间划分

不断积累的训练数据按天来划分，这样可以训练数据量的选择及更具灵活性，更重要的一点是，这样可以允许测试数据与训练数据分离且时间更新。由于上述做法考虑了诸如商品库存变化等实际效果，因此它对性能的评估更加实际。当旧的训练数据过时不再为模型所用时，上述按天来划分数据的做法也会带来好处。

按时间划分的做法也能允许以增量式的方法来构建训练数据，比如一次一天。这也意味着构建长期训练集的巨大开销可以分担到一段很长时间，这样就不会耽误每天的构建过程。

17.3.3 避免目标泄漏

在分类部分，我们至始至终都强调对特征提取进行精心规划以避免目标泄漏的重要性。为避免目标泄漏问题，Shop It To Me的工程师们使用了一个基于时间的多层数据隔离方案。对品牌偏好信息的SVD分解及k-means聚类基于早期的数据来进行，在模型建立时上述处理后保存的结果相对稳定。最后的簇模型应用到后期数据来产生实际的训练数据。此外，更近的数据放在训练算法之外来评估训练模型的精度。

17.3.4 调整学习算法

Mahout学习算法的默认行为可能并不能完全满足Shop It To Me公司的需要，这一部分是由于这里的分类器以推荐程序的方式来使用，另一部分是因为模型很难训练。上面的这些困难使得公司的工程师们对默认的Mahout Logistic回归算法在多方面进行了扩展。

1. 基于每个用户的AUC进行优化

`AdaptiveLogisticRegression`学习算法使用AUC作为默认的指标图来为二值模型调整超参数。在Shop It To Me分类系统中，AUC是一个度量哪对用户-商品可能会导致点击的精度指标。不幸的是，在点击模型中的一个最强信号涉及将用户分为点击和不点击两种。由于邮件营销系统的基本问题是对每个用户将商品排序，预测谁会点击并不是兴趣所在，最重要的是让商品展示给用户之后预测用户会点击其中哪些商品。为帮助学习算法找到解决正确问题的模型，需要一个不同于一般的指标图。

在Mahout中，有两个类可以用于计算不同形式的AUC。标准全局AUC可以通过`GlobalOnlineAuc`来计算，而基于组内排序的AUC则使用`GroupedOnlineAuc`来计算。`AdaptiveLogisticRegression`学习算法允许实现别的AUC和提供成组准则。对用户分组或者对用户聚类可以使得模型有别于简单选择点击用户的模型，从而达到良好的推荐效果。

2. 将排序和评分混合学习

到底是直接学习出概率还是学习出序是当前的一个主要研究领域。这些方法初看上去彼此等价，但是多位学者指出在实际情况下它们之间存在不同。一个最近的进展参见谷歌研究人员D.

17

Sculley的论文“Combined Regression and Ranking”。[1]Shop It To Me工程师通过对Mahout代码包装实现了这个技术，其中维护了训练样本的短暂历史数据，通过当前训练样本或当前样本和先前样本的差异梯度来更新模型。Mahout的`MixedGradient`类中有大量未经测试的上述技术的实现。

该技术对于Shop It To Me来说十分有效，但是要确定它真正正确的频繁程度需要更多的经验。

17.3.5　特征向量编码

图17-5给出了一个理想训练集中的一条记录，该训练集可能用于构建点击模型。在点击模型中，正如该记录一个更实际的版本一样，感兴趣的有三种类型的字段：

- 用户特有的字段；
- 商品特有的字段；
- 用户–商品的交互字段。



图17-5　一个简化的邮件销售系统点击模型中可能用到的训练记录字段。整条记录包含用户特有、商品特有和用户–商品的交互变量。这种类型的实际模型会比上述概念样例中的字段多很多

用户特有字段包括年龄组、性别、品牌偏好簇等变量的值，而商品特有字段则包括销售商、品牌和商品描述等变量的值。最有用的是用户–商品交互变量，这些交互变量使得模型不止是记录哪个或哪类商品最流行这样的信息。

与前面的特征编码例子相比，这里比较有趣的是交互变量，其涉及用户的品牌偏好簇编号（图中的`cluster`）、性别和商品品牌这些信息。

为使用推荐SGD算法采用的特征散列模式来对交互变量进行编码，编码中需要选择的向量位置要与交互变量各构成特征的位置相独立。例如，图17-6给出了我们想要的处理方式。假设我们手里有一个大小为100的稀疏向量，通过散列特征编码后特征“gender=female”可能被分配到位置8和21。类似地，”item=dress”可能分配到向量的77和87位置上。交互特征“female x dress”必须要分配到与上述四个位置尽可能统计独立的位置上去，但是同时该值仍然基于上述原始值来确定。

Shop It To Me工程师们完成上述位置统计独立性的一种做法是使用随机的种子来组合每两个位置。上例当中通过随机数生成器产生的种子为22和92。为得到交互特征的第i个位置，使用下式进行计算：

---

① D. Sculley，“Combined Regression and Ranking”，参见 http://www.eecs.tufts.edu/~dsculley/papers/combined-rankingand-regression.pdf，另见视频演讲：http://videolectures.net/kdd2010_sculley_crr/。

```
hash[i] = seed[0] * x[i] + seed[1] * y[i]
```

其中x和y分别包含“woman”和“dress”的特征位置。

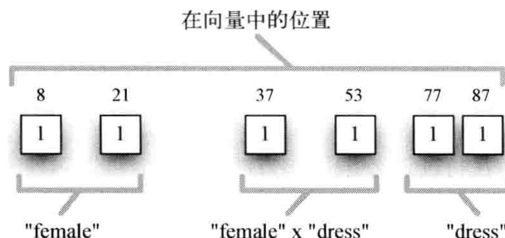


图17-6 交互特征需要散列到与其组成特征位置不同的位置

上述做法与布隆过滤器中使用的双散列密切相关，并且已有工作表明这种做法能够为组合特征提供相当好的独立的位置结果。代码清单17-1给出了上述思路下的一个交互特征编码器的具体实现。

代码清单17-1 交互特征编码器的一个实现代码示例

```
public class CategoryInteractionEncoder {
  private int[] seeds;                                    ❶ 种子组合散列值
  private CategoryFeatureEncoder[] encoders;              ❷ 编码器对成员变量进行编码
  private int probes = 2;
  CategoryInteractionEncoder(int seed, CategoryFeatureEncoder... enc) {
    Random r = new Random(seed);
    seeds = new int[enc.length];
    for (int i = 0; i < enc.length; i++) {
      seeds[i] = r.nextInt();                             ❸ 对种子进行确定性初始化
    }
    this.encoders = enc;
  }
  public void addToVector(int[] categories, double weight, Vector data) {
    int[] hashes = new int[categories.length];
    for (int i = 0; i < probes; i++) {                    组合散列值
      for (int j = 0; j < categories.length; j++) {
        hashes[i] += seeds[j] *
            encoders[j].hashForProbe(categories[j], i);   ❹ 组合每个特性的散列值
      }
    }
    int n = data.size();                                  利用组合散列值来设置值
    for (int h : hashes) {
      h = h % n;
      if (h < 0) {
        h += n;
      }
      data.setQuick(h, data.getQuick(h) + weight);
    }
  }
  ...
}
```

在交互特征编码器中，种子❶和编码器❷是常备对象。种子在构造函数中初始化❸，同时编码器队列也在构造函数中传入。在addToVector()方法中，交互特征中每个变量经编码器的散列值通过整数乘法❹运算来组合，该整数在交互特征编码器构建时基于当时提供的种子随机选择。这种乘法组合操作确保交互特征编码器所选择的任意特征位置都不可能与成员变量的特征位置相冲突。需要注意的是，CategoryFeatureEncoder是上述代码示例特有的类而并非Mahout本身的一部分。

也存在一些其他的对交互特征进行编码的方法。例如，可以对所有成员特征编码器保存第一和第二个散列位置的两个求和，然后利用标准的双散列算法。下面的代码给出了这种方法可能的一个实现过程：

```
public class CategoryInteractionDoubleHashingEncoder {
  private CategoryFeatureEncoder[] encoders;
  private int probes = 2;
  CategoryInteractionEncoder(CategoryFeatureEncoder... enc) {
    this.encoders = enc;
  }
  public void addToVector(int[] categories, double weight, Vector data) {
    int h0 = 0, h1 = 0;
    for (int j = 0; j < categories.length; j++) {
      h0 += encoders[j].hashForProbe(categories[j], 0);
      h1 += encoders[j].hashForProbe(categories[j], 1);
    }
    int n = data.size();
    for (int i = 0; i < probes; i++) {
      h = h0 + i * h1;
      if (h < 0) {
        h += n;
      }
      data.setQuick(h, data.getQuick(h) + weight);
    }
  }
  ...
}
```

❶ 对成员变量编码
❷ 计算两个基本的散列值
❸ 利用组合散列值来设置值

这里和前面一样对每个成员变量构建编码器❶,不同的是这里只使用成员变量的第一个和第二个探查值的求和结果❷。当计算交互特征的位置❸时，使用双散列对上面两个求和结果进行组合。

上面两种做法都没有进行严格的评估，但是两种方法在实际中看上去效果不错。然而具有讽刺意味的是，使用分类器来做推荐，用户和商品的交互变量是必需的，但是这些交互变量会增加变量编码和分类计算的工作量。由于Shop It To Me分类吞吐率要求相当高，分类系统中必须要进行加速处理，这一点至关重要。这些加速处理过程不但避免了编码的开销而且还避免了分类计算本身的计算开销。

17.4　加速分类过程

如前所述，Shop It To Me对最终系统中的分类器性能有永恒如一的需求，即要求分类达到惊人的速度。为了能在可用的不多的机器上实现每秒钟上千万的分类需求，必须要在基本的Mahout

之上进行改进。由于使用的内部特征向量超过100 000个元素，即使不考虑特征编码，采用原始的方法来达到所需的分类速度就需要每秒钟完成数千亿的浮点运算。要在仅仅几十个CPU核的情况下，要完成该任务不太现实。

为满足上述需求，Shop It To Me工程师们做出了如下改进：

- 将部分编码的特征向量放入缓存；
- 在编码器内部将散列位置值放入缓存；
- 将模型的计算过程划片来保证每个片能够分开被缓存。

上述改进措施最终将运行时的模型计算归结成3个查表和2个加法操作。必须的预处理包括每个用户一个编码运算及模型计算，每个商品上有几百这样的计算。上述修改带来的总体消耗节省量达到了3到4个数量级，这使得整个模型计算完全切实可行。这里列出的改进措施看上去有点简单，但是却提供了一个强有力的Mahout分类扩展方法。如果读者的项目当中所需要的吞吐率和Shop It To Me一样高，那么就可以考虑采用上述革新性方法。

17.4.1 特征向量的线性组合

Mahout中的特征提取系统将各个分开的特征向量相加得到总的特征向量。在Shop It To Me这个案例中，特征向量可以分成3部分，分别与用户、商品及两者同时相关。

$$x = x_{\text{用户}} + x_{(\text{品牌偏好簇}+\text{年龄组})\cdot\text{商品品牌}} + x_{\text{商品}}$$

由于可以将各部分放入缓存中而不需要每次在评估模型时立即计算它们的值，这种划分很明显有助于降低特征编码的开销。

由于整个特征向量很大，所以缓存它不是很起作用。而由于各独立部分的数目要小得多，所以部分缓存能起作用。基本上来说，大缓存所需的容量大概是多个小缓存容量的乘积。

图17-7给出的是在上述特征提取的3个部分如何用于模型得分计算的过程。

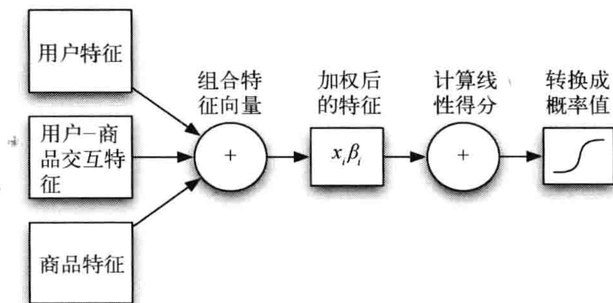


图17-7 Mahout SGD Logistic回归模型评分的流程

利用上述3个部分特征提取的缓存技术可以在某种程度上起到加速作用，但是单独来看，这种缓存技术导致的速度提升并不够。当然，它为其他的改进方法奠定了基础。

17

17.4.2　模型得分的线性扩展

将特征向量分成三部分能对特征编码起到加速作用，但是利用特征向量进行的模型计算在最后一步之前都是在进行线性操作。这能够允许使用更积极的缓存技术。此外，计算后面部分缓存的元素会小很多。

诸如Mahout SGD模型的Logistic回归模型的运行方式如下：首先对特征进行加权求和，然后将该值归一化到0~1之间。如果是对值排序，那么最后的归一化转换过程可以省略，因为它不会改变最终结果的顺序。在Mahout中，`AbstractVectorClassifier`中的`classifyScalarNoLink(Vector)`方法就正好省略了最后的转换过程。

一旦去掉最终的转换过程，模型就是一个简单的加权求和结果，那么就可以使用特征向量的一个线性组合。利用内积表示方法，模型的输出z可以写成：

$$z=\beta \cdot x$$

这里的β代表模型中每个特征的权重，而x代表模型中用到的特征。这两个符号代表的都是向量值。这里的乘号代表的是向量的内积计算，及将β和x对应元素相乘之后再求和。所有的密集计算都发生在这里的向量内积计算中。注意到上述公式可以按照特征向量划分的方式拆分成三部分：

$$z=\beta \cdot x_{\text{用户}}+\beta \cdot x_{(\text{品牌偏好簇}+\text{年龄组})\cdot \text{商品品牌}}+\beta \cdot x_{\text{商品}}$$

上述拆分过程可以通过修改图17-7生动地表示出来，图17-7给出的是Mahout SGD模型的计算流程，这样的话更多的计算就可以在最终组合之前分别完成。

图17-8给出了是用户、商品及用户–商品特征在得分组合之前分别应用于权重向量β的情况。这种方法的一个明智之处在于上述三个部分现在都能高效缓存。

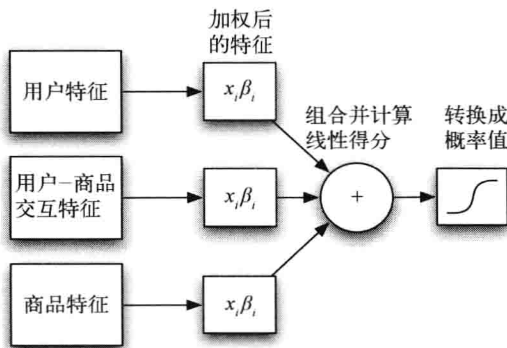


图17-8　在允许预计算和中间结果缓存的情况下如何重新组织模型评分的过程示意图

很重要的一点是，每部分的值现在都是单个浮点数而不是向量，因此在满命中率的情况下，对特定用户和商品的单个模型计算的开销就会从几千次浮点运算下降到仅仅两次查表和两次加法运算。由于得分的用户部分在内循环的外面计算，所以这里只有两次查表运算。而缓存小对象

也会减少内存开销，使得能够缓存更多的商品。

代码清单17-2给出了上述思路的实现代码。

代码清单17-2 使用缓存技术和部分模型评估来加速商品选择过程

```
public class ModelEvaluator {
  private OnlineLogisticRegression model;                    ❶ 从其他代码获得
  private List<Item> items = Lists.newArrayList();
  private Map<Item, Double> itemCache = Maps.newHashMap();
  private Map<Long, Double> interactionCache = Maps.newHashMap();
  private FeatureEncoder encoder = new FeatureEncoder();
  public List<ScoredItem> topItems(User u, int limit) {
    Vector userVector =
      new RandomAccessSparseVector(model.numFeatures());     ❷ 仅对用户进行一次评分
    encoder.addUserFeatures(u, userVector);
    double userScore = model.classifyScalarNoLink(userVector);
    PriorityQueue<ScoredItem> r =                           ❸ 累加最佳优先级队列
          new PriorityQueue<ScoredItem>();
    for (Item item : items) {                               查找缓存的商品得分
      Double itemScore = itemCache.get(item);
      if (itemScore == null) {
        Vector v = new RandomAccessSparseVector(model.numFeatures());
        encoder.addItemFeatures(item, v);
        itemScore = model.classifyScalarNoLink(v);          若未发现，则计算得分 ❹
        itemCache.put(item, itemScore);
      }                                                     ❺ 查找缓存的交互成分
      long code = encoder.interactionHash(u, item);
      Double interactionScore = interactionCache.get(code);
      if (interactionScore == null) {
        Vector v = new RandomAccessSparseVector(model.numFeatures());
        encoder.addInteractions(u, item, v);
        interactionScore = model.classifyScalarNoLink(v);
        interactionCache.put(code, interactionScore);
      }
      double score = userScore + itemScore                  ❻ 组合各部分
                  + interactionScore;
      r.add(new ScoredItem(score, item));
      while (r.size() > limit) {
        r.poll();
      }
    }
    return Lists.newArrayList(r);
  }
}
```

上面这个类给出了一个简化的代码片段，该片段可以通过商品和交互信息缓存来评估模型。假设在上述代码中，模型和商品列表通过其他方式传入❶。

该类的功能是，给定某个用户时首先计算该用户的得分的组成部分❷。然后扫描所有商品的列表并对每个商品计算模型得分。具有最高得分的那些商品被保留❸。模型得分需要计算三个部分。用户变量那一部分在商品循环之外计算❷并保留。然后在内循环，如果商品❹和用户–商品交互特征❺没有在适当的缓存数据结构中发现，那么它们会分别计算。FeatureEncoder是一个用

于为各种不同用户、商品及它们的交互特征的编码器进行包装的便利类。最后，各部分的得分组合成最终得分❻。

很明显，采用这种方式来拆分模型计算过程并不简单，但是有些情况下由于能够起到大幅度加速作用所以还是值得的。

17.5 小结

在本章给出的在线营销案例中，我们介绍了Shop It To Me的工程师们在开发Mahout分类器所用到的一些创新技术，他们利用这些技术来预测用户对看到的商品是否点击。他们使用Mahout分类器不管是在训练还是对真实数据运行部署所能达到的速度和规模展示了如何在大规模问题上应用Mahout。

这一章学到的有关分类的一些原则已经应用于Shop It To Me的实际系统中。一个重要的思路是，分类能够在一个排除传统推荐方法的系统中作为得到类似推荐结果的一种有效方法。

该案例阐示的另一个重要的原则是一开始对问题进行仔细分析很有价值，特别是关键点的考察相当重要，如果在工程设计中不考虑这些关键点可能会影响最终的成功。这些关键点常常来自极端的速度和规模需求，Shop It To Me这个在线营销系统的案例中给出了大量的上述挑战性问题。对问题进行分析可以让工程师意识到他们的系统切换到Mahout分类的价值所在，并且能够确定需要一些定制来扩展Mahout的效果。

Shop It To Me这个案例也展示了利用大规模联结来实现并行训练流水线的威力，这也是上一章中需要掌握的关键点。读者可能会发现，有很多应用当中都可以通过这种方法来构建流水线来满足超大项目的规模和速度的需要。

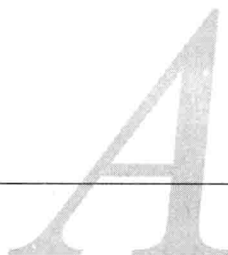
该案例中叶给出了在Shop It To Me系统中所用的革新性技术。这些新技术包括对象排序模型学习及元学习中的分组AUC使用。以排序和回归为组合目标的学习看上去对几乎所有的SGD学习应用十分有效。每当分类器用作推荐程序时，以分组AUC作为AdaptiveLogisticRegression的目标来优化元参数的学习看起来是一个十分有效的策略。

就吞吐率而言，这里的最大经验是缓存策略用于加速Mahout分类器的多种做法。这些方法包括部分编码特征向量的缓存、编码器内散列位置的缓存以及将模型计算拆分成可以各自独立缓存的片段。即使Shop It To Me在Mahout分类中使用了很多新技术，但是这绝不是事情的终点。训练速度的提高仍然有相当的空间，而且现有的部署系统不太可能压榨了所有的速度空间。

Mahout到底能走多远是下一章的主题，而这一章的创作就留给读者自己来完成。

附录A

JVM调优



这一附录讨论在基于Mahout的非分布式应用中，如何通过优化JVM配置来提升性能。这并不是Hadoop的优化指南，Hadoop的优化本身就是一个庞大的话题。这里的优化针对非分布式的Mahout，主要是关于推荐引擎的实现。尽管这些配置很可能对于任何基于Java的服务器端进程都会有效，但主要目标还是优化基于非分布式Mahout的推荐引擎。

当使用一个规模庞大的数据集时（可能是1千万甚至更多的偏好值），调优JVM配置来改善性能具有重要意义。与堆相关（内存相关）的配置是最为重要的。尽管最优配置取决于多方面因素，如操作系统、可用资源、架构以及JVM等，但下列JVM配置项将会给你一个较为理想的初始配置。

表A-1列出了Mahout相关的一些选项，除了`-XX:NewRatio`，其余选项在所有已知的Java 6 JVM中都是有效的，相关解释见下表。

表A-1　优化推荐系统涉及的JVM关键参数

参　　数	描　　述
`-Xmx`	设定Java允许使用的最大堆空间。例如`-Xmx512m`表示堆空间上限为512 MB
`-server`	现代JVM有两个重要标志：`-client`和`-server`，分别为客户端程序（运行时间短、占用资源少）和服务器端程序（长时间运行、资源密集型）选择合适的JVM配置。默认值取决于具体的JVM和环境。显然，在这里`-server`更合适
`-d32`和`-d64`	分别设定32位和64位模式。在一台64位的机器上，两种都是有效的。尽管通常情况下最好是让JVM自己决定，但选择32位模式可以降低内存需求。当然，32位模式下不可能使用超过2 ~ 3 GB的堆空间（具体取决于JVM），但是如果需求达不到这一界限的话，节省一些内存也不失为一个好的选择。在64位机器上选择32位模式会导致轻微的性能损失
`-XX:+NewRatio=`	有一部分堆空间是为生命周期很短的临时对象保留的，不能用于生命周期较长的数据结构。`Mahout`在运行时是有偏向的：它很少创建临时对象，而生命周期长的对象则需要消耗大量堆空间。默认情况下用于临时对象的堆空间比例太大，这显得有些浪费。此选项可控制用于临时对象的空间比例，例如，设为12时，只有1/12的堆空间用于保存临时对象。注意，此选项是Sun JVM所特有的
`-XX:+UseParallelGC` 和 `-XX:+UseParallelOldGC`	通过并行的垃圾收集机制使JVM更好地利用多个处理器或单一处理器的多个核（这在当前的桌面平台上都已经很普遍了）

为了演示这些JVM配置项的效果，我们用代码清单4-2中的代码进行了试验。首先从默认的32位客户端JVM开始：`-client -d32 -Xmx512m`。这些都是JVM标志，而非程序标志，所以它们需要出现在程序的类名之前。我们使用一个现代的64位计算机来进行测试，负载评估结果显示推荐时间为425 ms，稳定状态下需要消耗248 MB堆空间。（你自己的测试结果中时间和堆空间都有可能跟这里不同。）

如果我们将`-client`替换为`-server`呢？性能应该有所提高。测试结果显示，内存用量没有变化，但推荐时间降为192 ms。这表明为服务器端模式优化的JVM更适合此类应用。

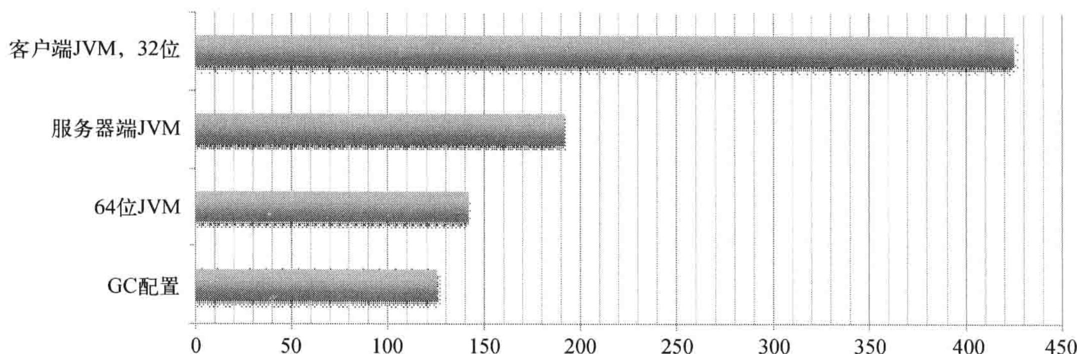
现在，将`-d32`改为`-d64`。你将遇到`OutOfMemoryError`错误。你可能需要分配768 MB的堆空间（`-Xmx768m`），然后再次运行。推荐速度再次提升，时间降至142 ms。这并不奇怪，在64位机器上64位模式效率更高。稳定状态下的内存需求则几乎没变：256 MB。从设计上讲，64位模式增加的内存需求用在对象和引用上，而与Mahout的推荐引擎创建的长时对象关系不大。

你可能注意到一个奇怪现象：既然稳定状态下的内存需求仅为256 MB，为什么在可用内存为512 MB的情况下，JVM还会因为堆空间不足而出错？在构建`DataModel`的in-memory数据表示过程中，内存需求会有一个峰值。

你可以尝试`-XX:+NewRatio=12`标志，它会将内存用量降至640 MB。

最后，试试加上`-XX:+UseParallelGC -XX:+UseParallelOldGC`。当可用的处理器核不止一个时，这会允许垃圾收集与主计算过程并行执行。在我们的测试机器上，推荐时间降至126 ms。与最初的425 ms比起来要好多了。

图A-1显示了在一个现代计算机上，不合适的JVM配置会导致推荐时间增长至原来的3.5倍。任何部署在严肃场合的推荐引擎都必须进行适当的优化。



图A-1　推荐时间随着JVM的优化而不断降低（单位：ms）

附录 B

Mahout数学基础

Mahout实现的很多算法都高度依赖于向量和矩阵相关的数学知识。Mahout自己的math模块就有自包含的向量和矩阵数学工具库，并且该模块可以脱离Mahout单独使用，它实际上是CERN的Colt库（http://dsd.lbl.gov/~hoschek/colt/）的一个改编版本。

本附录从实用的角度概述了Mahout math模块中的关键部分（在使用Mahout时，用户常常会遇到）。它们包括`Vector`和`Matrix`实例，以及相关的运算。这个附录并非十分完善的文档，读者感兴趣的话可以通过Mahout项目中的Javadoc文档和源代码进一步了解细节。

B.1 向量

尽管不同地方术语“向量”的意义基本类似，但在不同的上下文中其含义还是有细微差别的。大部分人都是首次在物理学中见到向量，在那里它表示方向并且通常用箭头展示。为达到目的，有时我们使用向量来代表空间中的点。这里的思路与以前并没有不同，只不过这里的点对应坐标原点到该点的向量。

在机器学习和Mahout中，很多时候是在一个更抽象的意义上使用向量，这时的向量并不一定对应某个几何上的解释。向量可以只是一个元组或者有序数值列表。向量有长度（或者称为维度），向量中从0到长度减1的每个索引（也叫位置）上有某个数值。向量通常写成诸如(2.3, 1.55, 0.0)之类的数值列表。例如，上面这个向量的长度为3，索引1位置上的值为1.55。在机器学习中，有时候处理的向量长度达到上百万级。

B.1.1 向量实现

`org.apache.mahout.math`中的`Vector`是一个应用时才实现的接口。该接口有多个实现。将向量以节省内存的方式表示，并且提供向量值的快速访问，对于Mahout来说相当重要。根据向量的本质和使用方式的不同，完成上述要求的最佳做法也不相同。

最重要的考虑因素是向量数据的*稀疏度*（sparseness）或*密集度*（denseness）。很多情况下，相对于向量长度来说，只有很少位置上的值才不为零，这也意味着其他位置上的值都为零。如果一个长度为1000的向量只有2个位置上的值不为零，那么和两个非零数值一起保存其余998个零不会有多大意义。当然，如果有100个位置上的数值为零，那么浪费的空间看起来就没那么大了。

如果相对于长度来说，有相对较多位置上的值不为零，向量称为密集向量。这种向量可以通过双浮点数数组保存数值来实现。此时向量的索引位置就直接对应数组的下标。密集向量的优势在于速度。由于按数组保存，因此访问和更新任意值都很快。`DenseVector`提供了这样一种实现。

稀疏向量要求一种不同的实现方式。为了不浪费空间存储零值，它只存储非零值的索引位置和对应值。我们可以使用散列表来实现从索引位置到值的简单映射。访问索引位置会比直接的数组访问要慢，但是也不会慢太多。`RandomAccessSpareseVector`是上述思想的一个实现。

基于散列的向量实现有一个问题，如果按索引位置顺序循环访问所有向量值，那么速度相对变慢，而按位置对向量遍历是一个比较频繁的需求。由于基于树结构的有序映射以及类似思路按序维护主键，所以它们可以解决这个问题，这种结构带来的代价是更长的访问时间。`SequentialAccessSparseVector`是第三种也是最后一种向量实现方式，它能满足上述要求。图B-1给出了密集向量和稀疏向量的例子。

密集向量	3.5	0.0	0.0	–1.2	2.0	0.0	–3.3
	0	1	2	3	4	5	6

稀疏向量	3.5	0.0	0.0	–1.2	2.0	0.0	–3.3
	0	1	2	3	4	5	6

图B-1　密集向量被表示成一个完整数组，因此只需要保存双浮点值。稀疏向量并不对零分量分配空间，因此需要保存一个整数来指出每个非零双浮点数值的索引位置。保存的值都用方框框住，其他值则很明确不需要保存

B.1.2　向量操作

下面介绍`Vector`中常用的方法。

`get(int)`和`set(int, double)`方法提供了对向量的基本操作，分别访问和修改索引位置上对应的向量值。`getQuick(int)`和`setQuick(int, double)`方法完成的功能与前面相同，但是没有提供边界检查功能，这有助于内部性能优化。其他调用程序或许应该坚持使用前两个方法。正如预期的那样，`size()`返回的是向量长度。

我们可以通过`iterator()`和`iterateNonZero()`完成对所有向量元素的遍历，前者访问所有索引位置的所有元素，而后者只访问非零元素。这些方法返回的`Iterator`对应`Element`对象，其中封装了索引位置和对应值。

这里实现的标准`clone()`方法用于`Vector`的复制，实现时也提供了构造函数的副本。`like()`方法返回一个同类的空`Vector`。这是一种工厂方法（factory method）。

最后，我们介绍math模块中涉及数学的地方。`plus(Vector)`和`minus(Vector)`方法提供了标准的向量加法和减法运算。我们也可以通过`times(double)`或`divide(double)`实现向量对一标量的乘法和除法运算。另请注意方法`dot(Vector)`和`cross(Vector)`，前者实现向量的内积（或点积）运算，后者创建了两个向量元素两两相乘得到的矩阵。

B.1.3　高级的向量方法

迄今为止提到的方法都像预期一样直接了当。Vector中的第一个不平凡方法是assign(Vector, DoubleDoubleFunction)。该方法对Vector进行修改，将其值设为该Vector和另一个Vector的值上的某个函数的结果。DoubleDoubleFunction封装了一个函数，该函数接受两个输入值，输出一个结果数值。这个方法与前面的基本方法没有本质区别，但可以让你的操作更高效。

例如，给定向量A和向量B，计算 $C = mA - B$ ，其中m是一个标量。这可以通过如代码清单B-1所示的myOperation方法来实现。

代码清单B-1　高效实现Vector运算

```
Vector myOperation(Vector A, Vector B, double m) {
  return A.times(m).minus(B);
}

Vector myFasterOperation(Vector A, Vector B, final double m) {
  A.assign(B, new DoubleDoubleFunction() {
    public double apply(double a, double b) {
      return a * m - b;
    }
  });
}
```

不论是myOperation还是myFastOperation都能完成上述任务。第一种方法很简单，但是却要比看起来慢一些。两次数学运算中的每一次都要分配一个全新的Vector对象，并且这些Vector也要遍历两次。然而，第二种方法并不分配新的Vector对象，它破坏性地更改A，这可能十分便利并如我们所愿。并且，它只需要对向量做一次遍历就可以计算出结果。

aggregate(DoubleDoubleFunction, DoubleFunction)方法能够简化基于向量所有元素值的函数的计算过程，特别当与Functions中现成的函数结合使用时。例如，要计算向量所有值的平方和，可以采用代码清单B-2中的任意一个方法来实现。

代码清单B-2　计算累积值

```
double mySumOfSquares(Vector A) {
  double sum = 0.0;
  for (Element e : A.iterateNonZero()) {
    sum += e.get() * e.get();
  }
  return sum;
}

double myOtherSumOfSquares(Vector A) {
  return A.aggregate(Functions.PLUS, Functions.SQUARE);
}
```

其中，在当前场景下，第二种实现方法效率要低一些，但是它要更紧凑一些。

B.2　矩阵

与向量不同，矩阵没有什么歧义，而且读者可能对它更熟悉一些，因为至少可能在数学课上看到过矩阵的运算。矩阵就是一张数值表。这种简单的构型被证明在物理学和线性代数中表示概念时十分有用，进而也扩展到了机器学习中。这些领域中矩阵运算是算法的基本组成部分，Mahout在多处都使用了矩阵。通常来说，在Mahout中将矩阵的行或列看成向量非常有用。

Mahout中的矩阵表示为接口Matrix的实现。其实现方式和Vector相同，即为不同场景或使用模式来进行相应优化从而得到多个类的Matrix实现。矩阵也有类似的稀疏性问题。SparseMatrix和DenseMatrix分别代表相对于规模来说非零元素很少和很多的矩阵。

SparseRowMatrix和SparseColumnMatrix是两个很有趣的SparseMatrix变种，分别应用于矩阵行或列常常当做一个访问单位（即一个向量）的情况。SparseRowMatrix应用于非零行很少，但是每行又必须作为向量来访问的情况。而SparseColumnMatrix则应用于相同的列向量情况。

矩阵操作

Vector中的大部分方法也在Matrix中存在，当然矩阵固有的二维性质使这些方法存在形式稍有不同。例如，get(int,int)方法返回的是特定行列号的矩阵元素值，而size()同时返回矩阵的行数和列数。like()方法和clone()方法的处理也以此类推。像类似assign(double)的便捷操作一样，这里支持对行列绑定标签。

Matrix分别提供了矩阵加法和乘法的实现：plus(Matrix)及times(Matrix)。其他常见的矩阵运算还包括矩阵的转置transpose()和行列式计算determinant()。

矩阵运算本身并没有什么令人惊讶之处，但是可以用一种紧凑的方式来实现原本复杂的运算。例如，代码清单B-3给出的是一个$m \times n$的矩阵，和一个n维向量（可以看成是一个$n \times 1$的矩阵）的乘积。

代码清单B-3　矩阵和向量的乘积

```
Vector vector = new SequentialAccessSparseVector(n);
Matrix matrix = new SparseMatrix(new int[] {m, n});
Matrix vectorAsMatrix = new SparseColumnMatrix(new int[] {n, 1});
vectorAsMatrix.assignColumn(0, vector);
Matrix productMatrix = matrix.times(vectorAsMatrix);
Vector product = productMatrix.getColumn(0);
```

上面这种运算常常在线性代数和机器学习中出现，从上面可以看到通过Mahout math模块实现这类运算十分简单。

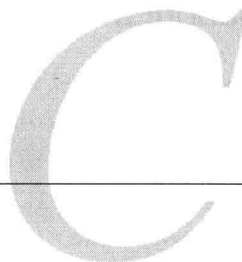
B.3 Mahout math和Hadoop

Matrix和Vector的实现常常用于Hadoop相关的Map和Reduce任务中。这也意味着它们必须可序列化，即可以转换为可存储、传输和重构的字节序列。Hadoop并没有Java的标准序列化机制java.io.Serializable来实现Matrix和Vector的序列化。实际上，它为此定义了一个十分类似的接口Writablel。为使自己能够以键–值方式在Hadoop中使用，类会实现Writable接口。

Vector和Matrix自己没有扩展Writable，因为这样做会使math模块依赖于Hadoop，而在概念上它并不依赖于Hadoop之类的系统。在核心的Mahout模块中，你可以发现VectorWritable和MatrixWritable，它们分别实现了Vector和Matrix的Writable接口。它们封装了关于如何在Hadoop中序列化向量或矩阵的知识。在读取或者写入向量或矩阵的Map和Reduce任务中，这些类中原始（raw）的接口实现要在传给Hadoop之前进行包装，返回时也是同样格式。

附录 C

相关资源



互联网总能够提供要多、更新的机器学习和Mahout相关资源；请使用你喜欢的搜索引擎搜索协同过滤（collaborative filtering）、聚类（clustering）、分类（classification）等主题。你也可以使用面向研究领域的搜索引擎，比如Google Scholar（http://scholar.google.com）。

这里给出了一些经典的论文和参考资料，以帮助大家进一步理解Mahout和本书的内容。

相关资源

- Anderson, M., M. Ball, H. Boley, S. Greene, N. Howse, D. Lemire, and S. McGrath. "RACOFI: A Rule-Applying Collaborative Filtering System." *Proceedings of COLA '03* (2003). www.daniel-lemire.com/fr/documents/publications/racofi_nrc.pdf.
- Blei, D.M., A.Y. Ng, and M.I. Jordan. "Latent Dirichlet Allocation." *Journal of Machine Learning Research 3* (2003): 993–1022. www.cs.princeton.edu/~blei/papers/BleiNgJordan2003.pdf.
- Breese, J.S., D. Heckerman, and C. Kadie. "Empirical Analysis of Predictive Algorithms for Collaborative Filtering." *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence* (1998). http://research.microsoft.com/research/pubs/view.aspx?tr_id=166.
- Dunning, T. "Recorded Step Directional Mutation for Faster Convergence." *Computing Research Repository* (2008): http://arxiv.org/abs/0803.3838.
- Gath, I., and A.B. Gev. "Unsupervised Optimal Fuzzy Clustering." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11 (1989): 773–780. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.2648&rep=rep1&type=pdf.
- Herlocker, J.L., J.A. Konstan, A. Borchers, and J. Riedl. "An Algorithmic Framework for Performing Collaborative Filtering." *Proceedings of the 22nd annual international ACM SIGIR Conference on Research and Development in Information Retrieval* (1999): 230–237. www.grouplens.org/papers/pdf/algs.pdf.
- Kannan, R., S. Vempala, and A. Vetta. "On Clusterings: Good, Bad and Spectral." *Journal of the ACM* 51 (2004): 497–515. doi: 10.1145/990308.990313.
- Lemire, D., and A. Maclachlan. "Slope One Predictors for Online Rating-Based Collaborative

Filtering." *Proceedings of SIAM Data Mining* (2005). www.daniel-lemire.com/fr/documents/publications/lemiremaclachlan_sdm05.pdf.

- Lloyd, S.P. "Least Squares Quantization in PCM." *IEEE Transactions on Information Theory* IT-28 (1982): 129–37. http://www.cs.nyu.edu/~roweis/csc2515-2006/readings/lloyd57.pdf.
- Lyon, C. "Movie Recommender." CSCI E-280 Final Project, Harvard University, 2004. http://materialobjects.com/cf/MovieRecommender.pdf.
- McCallum, A., K. Nigam, and L.H. Ungar. "Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching." *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2000): 169–178. www.kamalnigam.com/papers/canopy-kdd00.pdf.
- Murtagh, F. "A Survey of Recent Advances in Hierarchical Clustering Algorithms" *The Computer Journal* (1983): 345–359. http://comjnl.oxfordjournals.org/content/26/4/354.full.pdf.
- Ng, R.T., and J. Han. "Efficient and Effective Clustering Methods for Spatial Data Mining." *Proceedings of the 20th International Conference on Very Large Data Bases* (1994): 144–155. www.cis.temple.edu/~vasilis/Courses/CIS750/Papers/ClusteringSpatial-Ng.pdf.
- Olson, C.F. "Parallel Algorithms for Hierarchical Clustering." University of California at Berkeley, Berkeley, CA, 1994. www.eecs.berkeley.edu/Pubs/TechRpts/1994/CSD-94-786.pdf.
- Rajaraman, A., and J.D. Ullman. "Mining of Massive Data Sets," 2010. http://infolab.stanford.edu/~ullman/mmds.html.
- Rand, W.M. "Objective criteria for the evaluation of clustering methods." *Journal of the American Statistical Association* (1971): 846–850. doi: 10.2307/2284239.
- Rennie, J., Shih, L., Teevan, J., and Karger, D. "Tackling the Poor Assumptions of Naive Bayes Text Classifiers." *In Proceedings of the Twentieth International Conference on Machine Learning* (2003):616-623. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.322.
- Resnick, P., N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. "GroupLens: An Open Architecture for Collaborative Filtering of Netnews." *Proceedings of the 1994 ACM conference on Computer Supported Cooperative Work* (1994): 175–186, http://portal.acm.org/citation.cfm?id=192905.
- Sarwar, B., G. Karypis, J. Konstan, and J. Riedl. "Item-Based Collaborative Filtering Recommendation Algorithms." *Proceedings of the Tenth International Conference on the World Wide Web* (2001): 285–295. http://portal.acm.org/citation.cfm?id=372071.
- Sculley, D. "Combined Regression and Ranking." *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2010). www.eecs.tufts.edu/~dsculley/papers/combined-ranking-and-regression.pdf. See also the video lecture: http://videolectures.net/kdd2010_sculley_crr/.

索　　引

A

B

C

D

F

G

H

J

K

L